Validation of software measures[1] for the functional programming language Erlang

using the example of an instant messaging system

Diploma thesis
in fulfillment of the requirements for the academic grade of Diplom-Informatiker

submitted to the Department of Computer Science
at Humboldt-Universität zu Berlin

by Daniel Warmuth[2]

Supervisors:	Prof. Dr.-Ing. Beate Meffert
	Dr.-Ing. Robby Rochlitzer, ESYS GmbH
	Dr.-Ing. Frank Winkler
submitted on:	12 May 2012
defended on:	14 June 2012

[1] In the German original of this thesis, the term "metric" was used for the title and had to stay that way for academic/legal reasons. It had already been replaced by the more appropriate term "measure" for rest of the text. See **??** , p. 5.

[2] Translation kindly done by Jill Vyse, Berlin, in Summer 2017, finally edited by the author in October 2018. Notes and remarks in the German version, which were only concerned with explaining issues of translation ***into*** German, have been omitted in the course.

Contents

Remarks

The term "metric" used in the title will be replaced elsewhere in this work by the more appropriate term "measure" (see section 2.1.2.3, p. 19).

References to sources are given as follows: [i:p], where i is the index for the source in the bibliography and p is the page number in the source.

When subject-specific or specialized terms are defined or first introduced in this work, they are *slanted*; when they are used again they are generally indicated by a prefixed "·". These terms can be looked up in the index on page 132. The names of languages, products or measures appear in Small Caps, other points of emphasis in ***semibold italics***. Program code is set in `monospace` font.

All the statistical results presented below are significant to the *selected level of significance* [94:116], unless otherwise indicated. The selected level of significance is indicated with α, the *observed level of significance* [94:120] with p. The selected level of significance applies to the whole of the respective following section, unless otherwise indicated. The probability of a *type II error* [94:118] is indicated with β, where appropriate. When correlations are classified as "strong", "weak" or similar in the results of other studies, this is adopted from the authors of those studies.

1 Introduction

This study deals with the empirical testing of measures for the properties of software. In the first chapter, the problem of measuring software and carrying out the necessary validation is presented, similar studies by other authors are summarized and the contribution of this study is outlined. Then follows an overview of the structure of the rest of the thesis.

1.1 Presentation of the problem

The development of a comprehensive software system[3] is a complex undertaking: the requirements arising from the area of application have to be specified and the structure and behaviour of the software must be modelled, implemented and tested. The field of *software engineering* is concerned with developing methods which make it possible to master these processes with maximum effectiveness and efficiency. The aim is to develop high quality software through high quality development processes. ·Software measurement aims to quantify and to test objectively the extent to which these aims are achieved.

The measures of software measurement should identify specific properties of the development processes and products which are relevant for the successful development of the software. In particular, the attempt is made to draw from internal properties – which can be identified before and independent of the real, productive use – conclusions about external properties which will only be evident at a later point in time in interaction with the environment (see fig. 1.1 on the following page). Some of the interesting aspects of implemented program code are: How easy or difficult is it for software developers to understand the code? How susceptible to errors is the code? How complicated is it to test the code and rectify the errors? Reliable statements about these properties help to monitor and plan the development process. For this reason, it is necessary to check which measures permit statements about which properties. For this ·validation, the development methods must also be taken into consideration, as they also influence the properties to be measured. One factor here is the programming language used. While many studies exist for imperative, that is, procedural or object-oriented programming languages, there are only a few for functional programming languages [80:34].

[3] "System", "product", "software" and "program" are used as synonyms in this thesis.

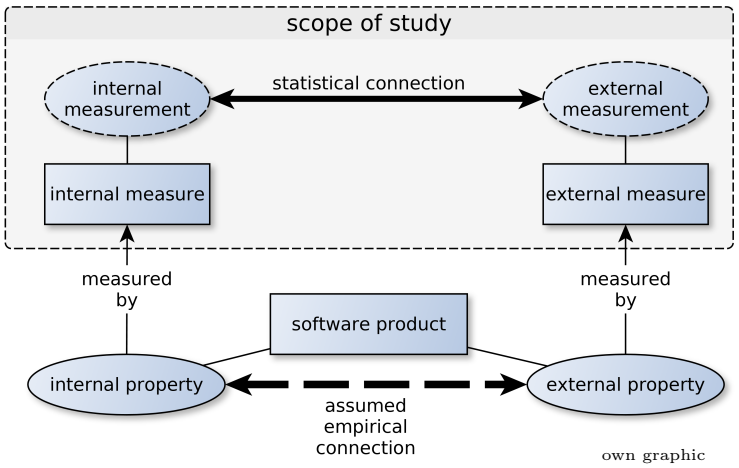


Figure 1.1 – Software measures aim at capturing internal and external properties, in particular of software products. The relations between measurement values should allow drawing conclusions about the relations between empirical properties. The appropriateness of these conclusions is examined through validation.

Therefore, this study will offer an ·empirical validation of selected measures on a comprehensive software system, which is implemented in the language ERLANG. To this end, the obtained measurements will be investigated in relation to the number of known errors in the system under observation.

1.2 Related studies

After intensive research of German- and English-language publications, the only empirical studies of software measures in the area of functional programming languages found were the following: the dissertations by Berg [6] and Ryder [80] and an article by Király and Kitlei [53][4][5]. These will now be briefly presented.

1.2.0.1 Berg – Connections with readability and comprehensibility

Starting from the question whether students who learn functional programming write "better" programs than those who learn imperative programming, Berg [6] carries out three experiments to investigate measures for readability and comprehensibility. The programs under consideration are small ones written by students in the framework of the experiments, after a one-semester programming course.

[4] The research in Ryder [80] was published again in Ryder and Thompson [81].

[5] Király and Kitlei [53:280] refer to Ryder [80], Ryder and Thompson [81] and Berg [6]. Ryder [80:40] in turn mentions Berg [6] as the only study of software measurement in the area of functional programming found at that time.

The first experiment [6:38ff.] was intended to test the hypothesis that functional programming languages lead to more readable programs than imperative languages [6:34]. To this end, the relationship of the measures ·cyclomatic complexity and ·programming effort[6] to the readability of programs in the functional programming language MIRANDA are compared to those of PASCAL programs. The programs are ranked by experts on a scale of readability. For the PASCAL programs, there are high correlations with this ranking for both measures,[7] while there are no significant correlations for the MIRANDA programs.[8] The conclusiveness of the result is limited due to the small sample size of nine and eight programs and the extent of disagreement among experts regarding the readability evaluation.[9] The hypothesis that the two measures are suited to the evaluation of readability of MIRANDA programs is not confirmed.

In the second experiment [6:103ff.], Berg considers function type expressions in the language MIRANDA, analogous to $f : \mathbb{N} \to \mathbb{R}$ in mathematics. He investigates the relationships between a specially defined measure [6:102] (called m below) and the time which experimental subjects need to understand such expressions. "Understanding" is operationalized by having the experimental subjects define any suitable function for every type expression. The same 40 expressions are presented to a total of 16 experimental subjects. For m, seven (in)equations were established as axioms which should be valid for the ranking of the basic type expressions, that is, for basic types such as lists, tuples and functions of basic types. So, for instance, a list of numbers is ranked as more difficult to understand than numbers themselves. In the first part of the experiment, the times required for processing both sides of these (in)equations are calculated. The following hypotheses are tested: for the inequations, the processing times for both sides should be significantly different; for the equations, the difference should not be significant. About half of the hypotheses are confirmed; for one equation axiom, there is an unexpected significant difference, for the rest of the axioms the differences are consistent, but not significant.[10] In the second part of the experiment, the ranking according to m is compared with that of the measured times. The rank correlation found[11] cannot be evaluated because Berg unfortunately does not indicate a level of significance here, and neither when he repeats this experiment [6:111ff.,128ff.][12]

The third experiment [6:147ff.] was intended to find out whether structured programs can in fact be understood more easily and more surely than unstructured

[6] According to Halstead [40:46ff.].

[7] Rank correlations according to Spearman: 0.58 (cyclomatic complexity) and 0.90 (programming effort); α not given, but apparently $= 0.05$; $p \approx 0.05$ and $p \approx 0.00$, respectively.

[8] Rank correlations: 0.56 and 0.38; $p \approx 0.07$ and $p \approx 0.18$, respectively.

[9] Kendall's concordance coefficient: 0.50 for MIRANDA compared to 0.74 for PASCAL; $p \approx 0.00$.

[10] The Fisher t-test is applied; $\alpha = 0.05$ [6:105].

[11] Spearman correlation coefficient of 0.59.

[12] There: Pearson correlation coefficient of the values of the self-defined measure and the measured times of 0.80, Spearman correlation coefficient of 0.74.

ones, as various style guidelines suggest [6:142f.].[13] 94 experimental subjects are each presented with six functions which vary in size and structure: small/medium/large and structured/unstructured. For each function and a given input, the experimental subjects should give the result of the function. The time needed and whether the answer is correct are recorded. It emerges that answers for structured functions are given more quickly[14] and are more often correct[15]; that for larger functions, the answers are given more slowly[16]; and, surprisingly, that for the structured functions, correct answers are given more often for larger ones than for smaller ones.[17] With regard to the latter, Berg suspects that more care is possibly taken with larger functions than with small ones [6:163].

1.2.0.2 Ryder – Connection with the number of modifications

Ryder [80:91ff.] studies the connections between a series of software measures and the number of corrections which are made in the course of developing programs. Code modifications are classified manually according to whether they add new functionality or correct existing functionality. Modifications are counted as corrections if they are intended to remove errors or improve structure. Ryder interprets the number of corrections as a measure of susceptibility to error [80:91]. He does not deal with alternative interpretations, such as, for instance, an indication of good maintainability. Ryder interprets software measures which correlate positively with the number of corrections as measures of "subjective complexity" of code, that is, the difficulty of understanding or changing code [80:103].

Two programs from university research projects written in the functional programming language Haskell are studied: a game called Peg Solitaire and a prior version of the refactoring tool HaRe.[18] Peg Solitaire only has 900 lines of code and 150 modifications [80:94], with the effect that there are mostly no significant results. In the version of HaRe which is studied, there are some 5,000 lines and 450 modifications.[19] [20] For every function of the programs, Ryder calculates the maximum values of a series of software measures and the number of code corrections during the whole development period [81:40]. This results in a data series with the maximum values of all functions for each software measure. A further data series contains the numbers of corrections of these functions. The

[13] Details of the meaning of ·structured and ·unstructured are given in section 2.2.3.1, p. 25.
[14] The F statistic is applied [4:68ff.]; $p = 0.000$.
[15] The Q statistic according to Cochran is applied [72:376f.]; $p = 0.000$.
[16] The F statistic is applied [4:68ff.]; $p = 0.000$.
[17] $p = 0.000$ to 0.004
[18] HaRe – The Haskell Refactorer: `http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html`
[19] The results presented below are from the program HaRe unless indicated otherwise.
[20] Taking a corpus of 14 other programs with a total of some 60 000 lines of code, Ryder [80:96f.] studied connections between different software measures – without, however, considering external features (such as the number of corrections). This study will not be discussed further here, as it does not deal with a validation in the sense under consideration here (section 2.3, p. 27).

linear correlation between these data series ($\alpha = 0.05$ [80:103]) is studied. One might criticize that Ryder repeatedly draws conclusions from results which are clearly statistically insignificant [vgl. 80:115] or that he is surprised about their deviation from significant results [vgl. 80:112]. Such conclusions are not given here.

Since intuitively a series of the measures under consideration depends on the size of the program, it is not surprising that they also correlate positively with the number of corrections. The strongest correlations ($r \geq 0.5$) are observed for the following measures: distance between declaration and use of identifiers,[21] number of variables declared in patterns,[22] size of patterns,[23] nesting depth,[24] number of operands and operators,[25] and fan-out[26] of functions.[27]

Weaker correlations ($0.25 < r < 0.5$) result for the following measures: depth and edge-node-ratio of the call graphs,[28] [29] depth of the sub-call-graph of individual functions,[30] number of data type constructors in patterns,[31] number of placeholder variables in patterns,[32] size of strongly connected components in call graphs (only for non-trivial recursion),[33] number of execution paths.[34]

For measures of recursion, Ryder observes no significant correlation with corrections [80:135ff.].[35] He attributes this result to peculiarities of the programs studied [80:144], which only contain a few recursive functions compared with a larger program corpus and whose nature is such that various measures are equivalent or inapplicable [80:140ff.]. There is hardly any correlation for the fan-in[36] of

[21] The distance is measured by counting the visibility scopes between the declarations and the points of use. Sum of individual values for all points of use: $r = 0.632$, $p < 0.0001$, maximum of the individual values: $r = 0.6006$, $p < 0.0001$ [80:128], linear regression with sum and maximum value: $r = 0.6829$, $p < 0.0001$ [80:277]. Smaller correlations result if the following are counted: the declarations in these visibility scopes, ($r = 0.546$, $p < 0.0001$), the number of intermediate lines ($r = 0.5334$, $p < 0.0001$ for maximum value) and the intermediate nodes in the ·parse tree ($r = 0.54$, $p < 0.0001$ for maximum value) [80:265].

[22] $r = 0.5927$, $p < 0.0001$ [80:112]

[23] $r = 0.5423$, $p < 0.0001$ [80:119]

[24] $r = 0.4208$ up to $r = 0.5692$, $p < 0.0001$ [80:118]

[25] $r = 0.5795$ and $r = 0.558$, respectively [80:156], with $p < 0.0001$ [80:267]

[26] See section 2.2.2, p. 25.

[27] $r = 0.5723$, $p < 0.0001$ [80:148]

[28] See section 2.2.2, p. 24.

[29] $r = 0.4932$, $r = 0.4258$, with $p < 0.0001$ [80:150, 266]

[30] $r = 0.3285$, $p < 0.0001$ [80:150, 266]

[31] $r = 0.3645$, $p < 0.0001$ [80:114]

[32] $r = 0.3572$, $p < 0.0001$ [80:117]

[33] $r = 0.3446$, $p < 0.0001$ [80:147, 266] for Peg Solitaire, which contains non-trivial recursion; $r = 0.0699$, $p = 0.105$ for HaRe, which contains only trivial recursion.

[34] $r = 0.286$ [80:155], $p < 0.0001$ [80:267]

[35] The only very weak exception, when $\alpha = 0.10$, is the binary predicate recurrence: Peg Solitaire - $r = 0.1119$, $p = 0.0883$ [80:266].

[36] See section 2.2.2, p. 25.

functions.[37]

Ryder concludes that his results have limited reliability as they are based on only two programs and he proposes further studies [80:163, 258]. The obtained measurement results are mostly in the lower value range, which indicates that upper limits for acceptable values can be found [80:256f.]; Ryder did not have the time to do this.

1.2.0.3 Király and Kitlei – prototypical use of RefactorErl to measure software

The authors Király and Kitlei belong to the project group developing the tool RefactorErl (see section 3.3, p. 48) which is used in the present study. Király and Kitlei [53:275ff.] compare two versions of RefactorErl with regard to various measures and find that the later of the two versions is more extensive, which is expressed in higher values for the equivalent measures and in the fact that it takes noticeably longer to analyse the later version. Their study serves above all to demonstrate their tool for measuring software. Its content is not relevant here, because it is not a validation study.

1.3 Contribution of this study

The studies published to date on the validation of software measures for functional programming languages are based on data from small or medium-sized experimental software projects with no more than a few thousand lines of code. For this reason, the present study considers a larger software project which contains tens of thousands of lines of code and is in constant professional use: the widespread communication server ejabberd, which is implemented in the functional programming language Erlang. Various software measures are obtained and the relationships to external measures of quality are studied; these measures are derived from a database of known problems and from code modifications. On this basis, statements are made about the extent to which specific software measures can be used as indicators for specific properties of a program. Further, comparative values which may be useful for evaluating the measurement values for other systems are determined.

1.4 Structure

This thesis is divided into seven chapters and four appendices.

[37] $r = 0.0842$, $p = 0.0507$ [80:148]

In the above, the need for software measurement and for validation has been motivated and the contribution of this study in the context of related studies has been summarized.

In chapter 2 on the next page, the facets of software quality will be presented. Important concepts of measurement theory will be defined as the basis for software measurement. For the field of product measures, the aims and applications in the software development process will be explained and the modelling of software products for the purpose of measurement will be described. Methods of validation of software measures will be presented and finally significant aspects of functional programming and their effect on software measurement will be discussed.

In chapter 3, p. 40, the analytic tool RefactorErl and a selection of measures of internal and external quality features will be presented.

Chapter 4, p. 51, introduces the software product studied, ejabberd. After describing the structure of the study and the type of data studied, hypotheses concerning the connections between internal and external measures will be put forward and then statistically verified.

In chapter 5, p. 86, the propositions on the hypotheses will be discussed in summary.

Chapter 6, p. 91, is a summary of the study and chapter 7, p. 92, discusses open questions and possible extensions.

After the list of diagrams and tables and the bibliography, there are the following appendices: a translation of the Software Measurement Ontology (appendix A, p. 106), the documentation of the measurement and analysis environment (appendix B, p. 109) and supplementary material and tables on various chapters (appendices C and D).

2 Foundations

In the previous chapter, software measurement was motivated and an outline of the present thesis was given; this chapter now offers an explanation of the contribution which software measurement can make to the quality control of software. Some aspects of software quality, the fundamentals of measurement and software measurement will be introduced and the modelling of software products for the purpose of measuring them will be described. The procedures for the validation of software measures will be discussed. Finally, functional programming and its features in connection with software measurement will be elucidated.

2.1 Software measurement

> "The use of software metrics reduces subjectivity in the assessment and control of software quality by providing a quantitative basis for making decisions about software quality."
>
> *IEEE Standard for a Software Quality Metrics Methodology* [46:iii]

Software measurement is an aspect of *software engineering*, that is, the "application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software" [19:67]. Software measurement is concerned with quantitative measures for properties of the processes and products of software development. Correspondingly, a distinction is made between process measures and product measures [57:212]: *process measures* capture properties of the development process, such as productivity or efficiency; *product measures* represent properties of the software product, such as size, structure, error rate, runtime behaviour. Product measures may be recorded statically or dynamically, in various phases of the development process and at different levels of abstraction: statically for documents in the phases of specification, design and implementation, dynamically during execution. In the following section, only static product measures will be considered.

2.1.1 Software quality

ISO standard 25010 defines *software quality* as "the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides

value" [49:2].[38] For overall quality, a distinction is made between *product quality* and *data quality*, as well as *quality in use*. Product and data quality concern those properties of the software product and the processed data which are independent of their behaviour and are also described as *internal properties* [28:74]. Figure 2.1 presents the connections between these areas.

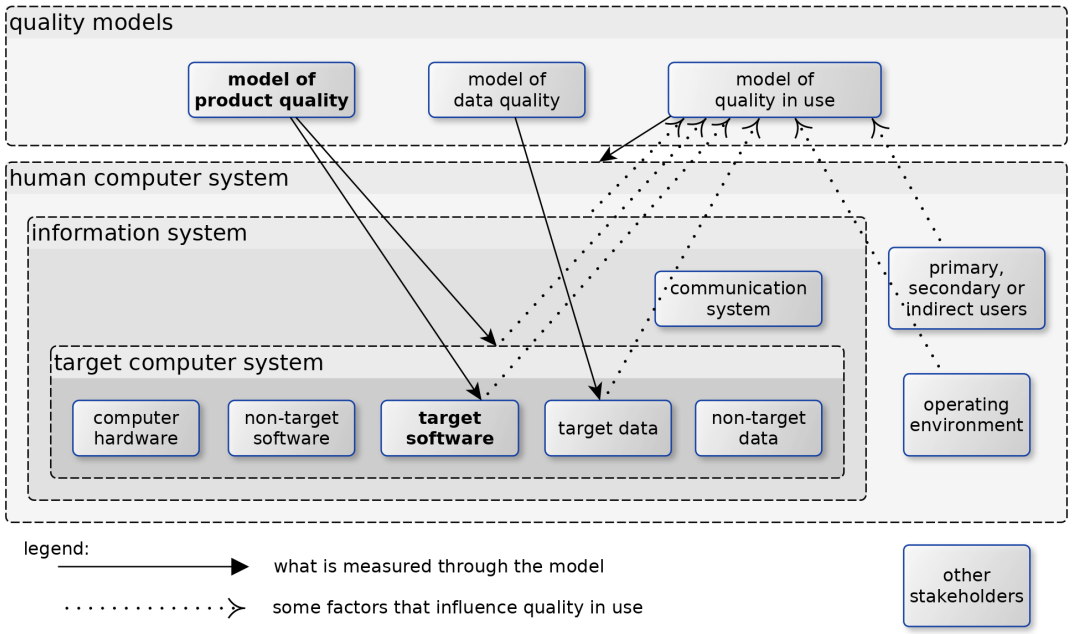


Figure 2.1 – Models for various aspects of software quality, and corresponding components of the human-computer-system according to [49:5]

Product quality, the measurement of which is the subject of this study, is modelled as a hierarchy of different properties, with more abstract properties being subdivided into more concrete sub-properties. The aim is to be able to record properties directly at the lowest level with (so-called *internal*) software measures. ISO 25010 uses eight properties with 32 sub-properties to describe product quality [49:3f,10ff.]: *functional suitability*, *reliability*, *performance efficiency*, *operability*, *security*, *compatibility*, *maintainability*, *portability* (see fig. 2.2 on the following page). One example of a model which in the end directly maps these sub-properties to software measures is the SIG Maintainability Model [42], which refers to the factor *maintainability* according to ISO 9126 [47]. This is defined by ISO [49:14] as the "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers", where "modi-

[38] The ISO/IEC standard 25010 is the successor to ISO/EC Standard 9126, which it replaced in March 2011 [49:v]. The German version of ISO/IEC 9126 was the DIN Norm 66272 [25:22ff.], which was, however, completely withdrawn (see `http://www.beuth.de/de/norm/din-667272/2385241`). A German version of the guideline for the whole ISO Standard Series 250xx is in preparation (as of 9 May 2012, see `http://www.nia.din.de/projekte/norm/DIN+ISO%2FIEC+25000/de/136443377.html`).

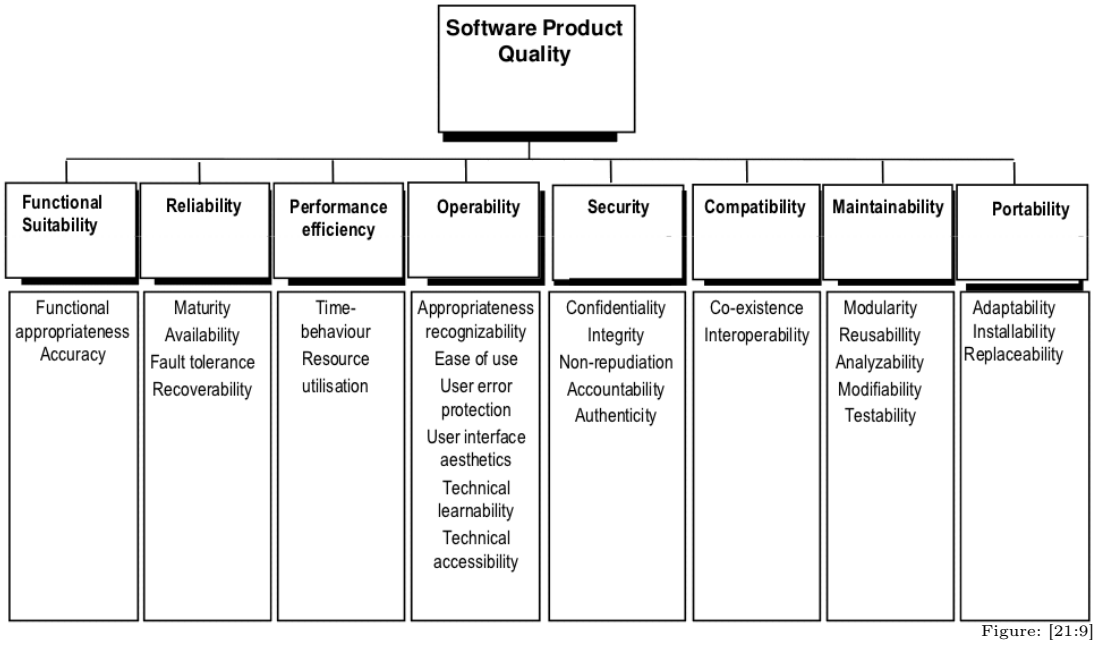


Figure 2.2 – Quality model for software products according to ISO/IEC 25010 [49]

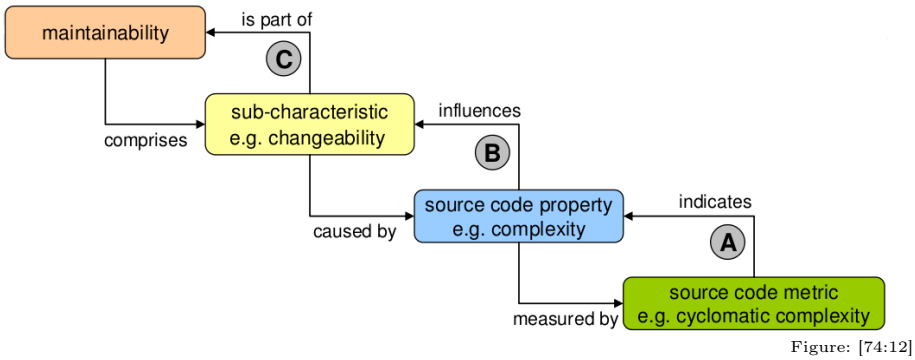


Figure 2.3 – Excerpt of the SIG Maintainability Model [42]

fications can include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications."

"Complexity"

The literature about software measurement often refers to "complexity", without a definition of its meaning. This vague concept is dealt with as an individual property and is said to be responsible for a wide range of effects: for a large number of errors in the code; for complex testing procedures; for difficulties in restructuring software, and so on. This is combined with an effort to try to express this ***complex*** phenomenon, "complexity", in a single number. (Similar attempts

were made to express "intelligence" in terms of a single "intelligence quotient" [35].) Zuse [99:31] quotes Howatt and Baker [45] on this:

> "Measures of software properties should not be combined into a single-valued measure. One number cannot convey the information that a set of individual measures can; information is lost. We therefore propose that individual measures be made components of a vector of measures. This will provide complete information on each of the individual properties."

Apart from objective properties, the aspects of software quality to which "complexity" is attributed also depend on human capabilities, unlike computational complexity. A program with a high computational complexity may be easy to understand (for example, Bubblesort), that is, may have low "complexity"; a program which is difficult to understand – "more complex" – may, by contrast, be more efficient (for example, Quicksort). Thus, Zuse [99:1] also speaks of "psychological complexity": "The true meaning of the term software complexity is the difficulty to maintain, change and understand software. It deals with the psychological complexity of programs."

2.1.2 Basic concepts of software measurement

2.1.2.1 Foundations of measurement theory

In this study, *measurement* is understood in the sense of *representational measurement theory* [95:168]: "Measurement assigns numbers to objects or events, provided that this assignation is a homomorphous transformation of an empirical relative into a numerical relative." [75:138] The terms *relational system*[39] and *measure* are defined below. A more extensive introduction to this theory of measurement can be found in Bortz and Schuster [11:15f.].

Relational system A *relational system* is a tuple $(A, R_1, \ldots, R_n)$, where A is a non-empty set of objects and R_i $(i = 1, \ldots, n)$ are n-ary relations over A [99:40]. If the relations include closed binary operations on the elements of A, these will be indicated separately by $\circ_j$ $(j = 1, \ldots, m)$.

A distinction is made between empirical and formal relational systems. The objects of an *empirical relational system* are, for instance, program texts, and the relations are "as easy to understand" or "easier to understand", for example. An empirical binary operation is the combination of two program texts. The objects of a *formal relational system* are usually numbers with such algebraic relations as $\geq$, and operations such as addition and multiplication.

[39] In this study, the term *relational system* will be used instead of its synonym, "relative".

Measure Let $\mathbf{E} = (E, R_1, \ldots, R_n, \circ_1, \ldots, \circ_m)$ be an empirical relational system, $\mathbf{F} = (F, S_1, \ldots, S_n, \bullet_1, \ldots, \bullet_m)$ a formal relational system, and μ a mapping from E into F. μ is a *measure* (with respect to $\mathbf{E}$ and $\mathbf{F}$), if and only if it is a homomorphous mapping of E into F [97:16]. In this context, homomorphous means that the relations of the measurement values correspond to the relations of the empirical objects, i.e. that for all i, j and all a, b, with $a_{i_1}, \ldots, a_{i_k} \in E$ it holds [99:40f.] [cf. 11:16]:

$$R_i(a_{i_1}, \ldots, a_{i_k}) \Leftrightarrow S_i(\mu(a_{i_1}), \ldots, \mu(a_{i_k}))$$

and

$$\mu(a \circ_j b) = \mu(a) \bullet_j \mu(b)$$

Scale If μ is a measure with respect to the relational systems $\mathbf{E}$ and $\mathbf{F}$, the triple $(\mathbf{E}, \mathbf{F}, \mu)$ is also called a *scale*. Depending on the type of relations from $\mathbf{E}$ which μ retains in $\mathbf{F}$, it forms one of the *scale types* used in statistics (including nominal scale, ratio scale, interval scale, relational scale and absolute scale [89:6f.]). For a large number of software measures, Zuse [99] derives the type of scale defined by them theoretically with respect to the so-called "subjective complexity".

2.1.2.2 A uniform vocabulary

On the basis of a systematic comparison of international standards and research publications on software measurement, García et al. [32:631] come to the conclusion that the field of software measurement "is currently in the phase in which terminology, principles and methods are still being defined, consolidated, and agreed." They found serious cases of homonymy and synonymy, and there were disparities and gaps, even in some fundamental concepts [32:635]. Some of these kinds of problems will be dealt with at the end of this section (section 2.1.2.3, p. 19).

By synthesizing and completing the existing terminology, García et al. developed an *informal ontology* [56:274], which is intended to contribute to standardizing the terminology as a structured, controlled vocabulary [56:280]: the so-called Software Measurement Ontology [32:635ff.] [7:175ff.] (see fig. 2.4 on the following page and appendix A, p. 106, for the author's translation [into German]). The terms used in this study will now be defined according to this ontology, just with the definitions for ·measure and ·scale replaced by the above, in order to be consistent with representational measurement theory.

Measurement Totality of the operations to calculate a ·measurement value for a specific attribute of a ·measurement object by means of a ·measurement approach.

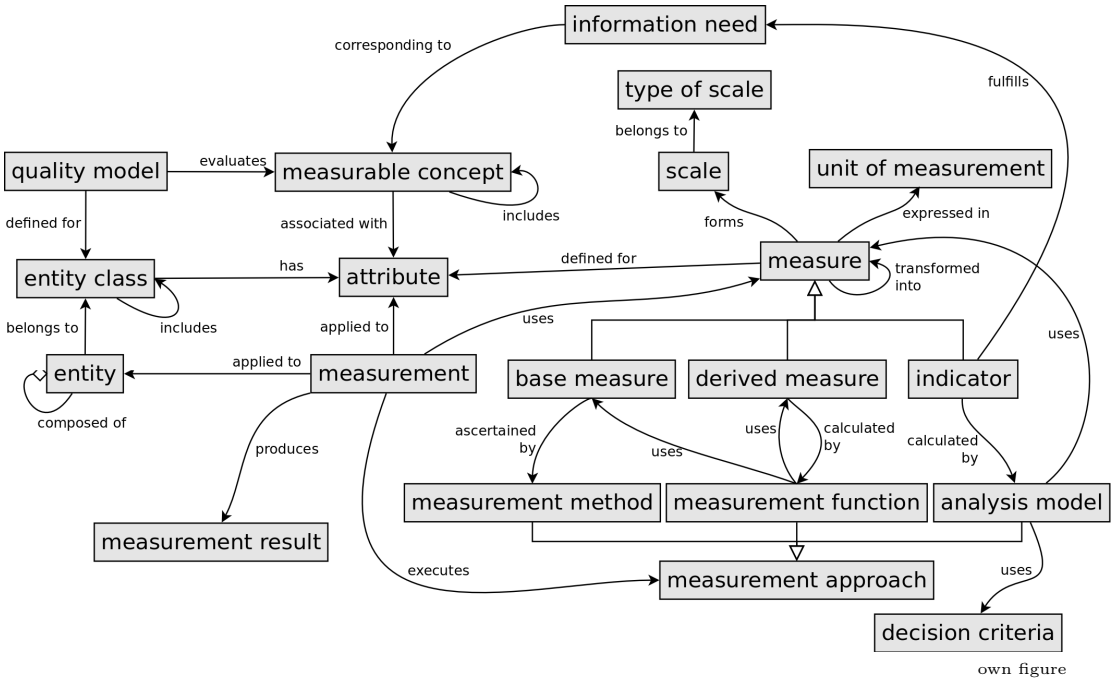


Figure 2.4 – Adapted Software Measurement Ontology [cf. 7:181]

Measurement approach A *measurement approach* is a series of operations which aim to establish the value of a measurement result. *Measuring method* and *measuring function* are types of measurement approaches:

Measuring method Logical series of operations, described in general terms, which are used to quantify an attribute with reference to a particular ·scale.

Measuring function Algorithm or calculation combining several ·base measures or ·derived measures.

Measurement value The number or category which is assigned to an attribute of a ·measurement object by a ·measurement.

Measurement object Empirical object which is to be characterized by the ·measurement of its attributes.

Measure Homomorphous mapping μ from the set of empirical objects E into the set of formal objects F. For $e \in E$, $\mu(e) \in F$ is called the ·measurement value for e. An established ·measurement approach serves to implement in practice this theoretical mapping. A distinction is made between base measure and derived measure:

Base measure A ·measure of an attribute which is not based on any other ·measure and whose ·measurement approach is a ·measuring method.

Derived measure A ·measure which is formed from other ·base measures or ·derived measures using a ·measuring function as the ·measurement approach.

The term *internal measures* is used for ·internal quality properties, and the term *external measure* is used for ·external quality properties. One example of an internal measure is the number of lines in a program, while the failure rate is an external measure.

2.1.2.3 Ambiguous terms

The term "metric" The literature on software measurement often refers to "metrics" instead of ·measures (see, for instance, Balzert [5]). This term is confusing, because in mathematics, *metric* usually denotes an interval function for points in a space [33:766]. However, measurement values in the above sense do not denote intervals and meaningful intervals between measured values can only be given under specific conditions. For this reason, Zuse [99], Liggesmeyer [57] and others reject the term "metric" and use instead the term ·measure.

The term "measure" There is a difference between the term ·measure in representational measurement theory and the term measure in mathematical measure theory. In the latter, a *measure* is a mapping from the set of a σ-algebra into the non-negative real numbers, such that, inter alia, the sum of the images of disjunctive sets equals the image of the union of these sets [3:17]. Krantz et al. [55:199ff.] discuss how probability measures, which are based on this measure theory, can be dealt with in terms of representational measurement theory.

In [99:29], Zuse describes as a measure ***every*** mapping μ from the set of empirical objects E into the set of formal objects G; this would be problematic, because then not every application of a "measure" would be a ·measurement. However, in keeping with his efforts over many years to place software measurement on a solid foundation of measurement theory, he uses the above definition in [97:16] and [98:3].

Standard works on representational measurement theory seldom use the term "measure"; Krantz et al. [55], Orth [75] and Bortz and Schuster [11] speak of ·scales in the above sense, without explicitly calling the homomorphous mapping which belongs to a scale a ·measure.

The term "scale" "Scale" is often used as a mathematical term, but it is seldom defined. According to Walz [93:39], the term "scale" only refers to the ***range*** of a ·measure in the above sense. Correspondingly, a "scale" would only be a set of numbers which as such say nothing about specific relations. In the SOFTWARE

Measurement Ontology, too, "scale" and "type of scale" are defined in this sense [32:636].

At first sight, DIN [24:26] is in accordance with this when it describes a "scale of quantity values" as a "set of values of quantities". However, values are defined there as "numerical value ***and reference***" [emphasis added] to a unit of measurement, a measuring procedure or a reference material [24:23]. This means that it is not simply a question of numbers, but of numbers with a specific empirical meaning. In the definition of a ·scale, the same is just more clearly expressed as a homomorphism between the empirical and the formal system.

2.1.3 Applications of static product measures

> "You cannot understand the beauty of a painting by measuring its frame or understand the depth of a poem by counting the lines."
>
> Marinescu and Lanza [60:46]

Software measurement is one of many methods of assessing and testing software quality. Some other ways are formal specification and verification, and testing (see Schlingloff [83:341ff.] for a brief overview). The expressiveness of these methods and the effort involved vary. With formal methods, specific properties of a program can be ***proven*** – however, in general these procedures require a lot of effort and are difficult to automate, so they can only be used for especially important systems. In testing, the complexity can be reduced by the selection of the testing strategy and (semi-)automatic test case generation, although the resulting statements are weaker than with formal methods, because: "Program testing can be used to show the presence of bugs, but never to show their absence!" [22:1]. However, with increasing test coverage there is a reduced likelihood that errors remain undetected.

The statements resulting from software measurement are even "weaker" because they often merely point to ***possible*** weak points in a program. Furthermore, the importance of software measures has received the least scientific investigation. Software measures have the advantage that they can be determined with little effort and entirely automatically.

Static product measures for implementation documents, such as those examined in this study, are generally expected to quantify how difficult it is to understand, alter or test a segment of code [80:11]. Balzert [5:232] emphasizes that software measures have to be treated with caution. As the software development process is not yet entirely understood, hypotheses about connections between software measures and interesting properties do not yet form a secure base for quantitative statements. One aim of validating software measures is to reduce this insecurity.

Software measures are used to assess and compare the current quality of software and to predict future quality [84:412]. From the current or predictive quality evaluation, refactorings can be derived [60].

2.1.3.1 Evaluation

Software measures can be used to establish requirements of software quality quantitatively, in order to be able to monitor changes of software quality during and after the software's development, and to test the fulfillment of quality requirements [46:1]. Various systems or parts of systems can be compared in order to make a selection or to assign development expenditure [86:326].

In each case, software measures abstract from many aspects of software, in order to be able to quantify one or a few aspects. For this reason, one individual software measure never assesses a software system as a whole. In order to obtain comprehensive results, a number of suitably selected measures have to be evaluated [5:478]. Here, useful summaries of various measures (for instance, the average number of lines per class) can sometimes make properties easier to recognize than the individual values [60:23].

In the application of software measures, it is important to note that outlier values occur frequently. But if a component manifests extreme values on various measures, it should be investigated more closely and possibly be modified [86:341] (see section 2.1.3.3 on the following page).

2.1.3.2 Prediction

Software measures can be used to predict properties which will only become measurable later, for instance, the error rate of programs [57:231]. On the basis of these predictions, the anticipated quality can be evaluated at an early stage, and preventive measures to reexamine or modify it be initiated [84:412]. The models which have been investigated in the literature are mainly those intended to predict the software's proneness to errors.[40] In this study, predictive models will not be considered in detail, although the validation of software measures creates a basis for the development of predictive models from these measures [39:3].

[40] Fenton and Neil [27] offer a critical overview of such predictive models up to 1999 inclusive; Hall et al. [39] present a systematic overview of the literature regarding research in the years 2000 to 2010.

2.1.3.3 Refactoring

The restructuring of a software system with the aim of improving the internal structure without changing the external functionality is called *refactoring* [30:xiii]. On the one hand, there are schematic guidelines for the design of software systems – so-called *design patterns* – which are supposed to lead to higher software quality [31]. On the other hand, patterns in the structure of software systems are described which should be ***avoided*** because they negatively affect the software quality. Fowler [30:67ff.] speaks of *bad smells in code*, Brown et al. [14] speak of "antipatterns". One example is when most of the functionality of an object-oriented software system is centralized in one single class, instead of being distributed equally across classes with clearly delimited responsibilities (Large Class [30:71]). This is considered to prevent the reusability and comprehensibility of this class and of the whole system [60:80].

In the first instance, software measures can only show the strength of a specific attribute (for example, maintainability) in the software. This generally does not show directly how the software can be improved [vgl. 80:16]. For example, Spinellis [86:331f.] points to the fact that the ·maintainability index can be increased ("improved") in trivial but useless ways by introducing a comment before each function, giving the name of the function and its parameters. However, this would not improve the maintainability of the code and might, on the contrary, make it worse. Therefore, Spinellis [86] recommends that one should not randomly attempt to maximise measurements (here, of the maintainability index), but should use them as clues to potentially problematic parts in the code, which need to be examined. Liggesmeyer [57:239] warns against basing decisions, such as those concerning the decomposition of modules, on a single measure, as this only measures a single aspect of quality.

Marinescu and Lanza [60] show how software measures can be used for the automatic recognition of potential weak spots in the design of object-oriented software systems. To this end, they present rules which are supposed to point to patterns of poor design or ·bad smelling code by linking various software measures [60:49], as the starting point for a manual check of the affected components [60:56f.]. These rules present models showing which aspects of software quality can be determined by which measures. Figure 2.5 on the next page shows the graphic representation of the rules for recognizing the above-described pattern ·Large Class, which Marinescu and Lanza [60:80] call "God Class".

2.1.4 Limits of static product measures

Various factors reduce the expressiveness of static software measures. For dynamic properties, such as speed or memory footprint, static analysis can only offer approximate limits. On the other hand, static analysis methods can also make

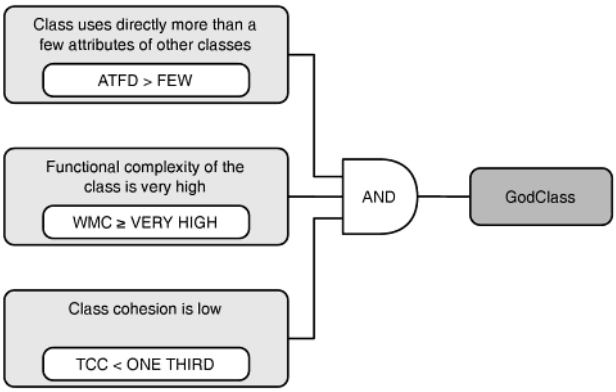


Figure 2.5 – Detection rule for "God Class" by Marinescu and Lanza [60:81], with informal conditions and their mapping to three software measures

semantically more meaningful statements than software measures. For example, every compiler can recognize and name real errors. Since software measures are more abstract, they can give summary information about very large systems in order to obtain an overview, which would not be possible with detailed information, such as that about individual errors. Furthermore, with product measures the hope exists that weak spots in programs can be identified ***before*** errors occur in them.

When interpreting product measures, attention must be paid to the fact that there are many other influences on software quality, apart from those measured, which have to be taken into consideration (see section 2.3.4, p. 31). These include properties of other products, such as specification and design documents; of the development process, such as the working hours expended, the number of developers, the methods applied; and finally, the "human factor", such as the qualifications and motivation of the developers. For example, some investigations which have shown that larger programs contain proportionally fewer errors [44:9f.] or are better understood [6:163], explain this result by the greater care taken by the developers.

2.2 The modelling of programs

Software products are modelled in various ways in order to measure them. The entities under consideration are therefore not the software products themselves, but their models. In the context of psychology, Gigerenzer [34:60] speaks of "measuring as modelling", emphasizing that the models measured are not identical to the real objects. When interpreting software measures, one has to take into consideration the empirical properties from which the model in question has abstracted.

At the lowest level of abstraction, programs in most programming languages consist of a series of characters by which a series of symbols in the programming language are coded. Some measures start directly from the textual form, abstracting from the semantic content of the code: number of characters, number of lines, length of the lines (see fig. 2.6).

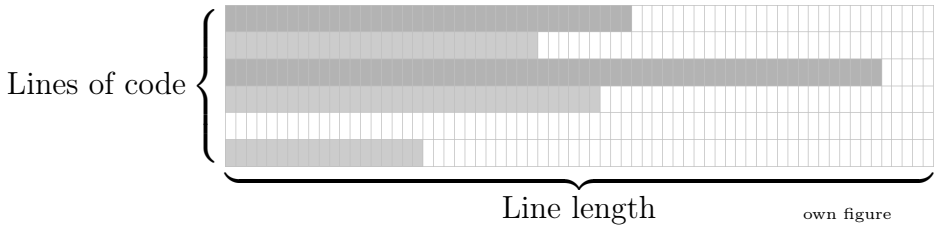


Figure 2.6 – Textual form of a program

Other measures require a lexical analysis, in order to decode the symbols, and frequently also a syntactic analysis, in order to establish the grammatical structure [2:6f.]; this includes, for example, the number of comment lines.

In turn, other measures refer to the call relations between functions, to the import relations between modules or to the so-called control flow (sections 2.2.2 and 2.2.3). Their modelling is based on the abstract syntax tree, which will be presented now.

2.2.1 Parse tree and (abstract) syntax tree

A *parse tree* (also called *concrete syntax tree*) represents the syntactic structure of a word in a language, that is, the derivation of this word by means of a (context-free) grammar [2:36]. The root node corresponds to the start symbol of the grammar, every internal node to a non-terminal symbol and every leaf to a terminal symbol or the empty word ϵ.

An *abstract syntax tree* (also abbreviated as *syntax tree* or *AST*) abstracts from syntactic elements which are not significant for the meaning of a word [2:60]. Thus, for example, operators and key words are not presented as leaves but as internal nodes, with their operands as child nodes [2:351].

2.2.2 Call graph

A (static) *call graph* is a directed graph which represents which functions can potentially be directly called by which functions [41:4].

Every node represents a function. An edge (F_1, F_2) means that the function represented by node F_1 can be directly called by the function represented by node F_2 [41:12].

Analogous to this, a directed graph can be defined to represent which modules import which other modules, that is, give direct access to the functions contained in them.

The number of edges which originate from the node of a function in the call graph can be called the *fan-out* of this function, the number of edges which lead to the node of a function in the call graph can be called the *fan-in* of this function.

2.2.3 Control flow graph

A *control flow graph* (abbreviated as CFG) is a directed graph which represents the potential program execution sequence, the so-called *control flow*, within a function [41:4].

Every node represents a *basic block*, that is, "a series of consecutive instructions which the control flow enters at the beginning and leaves at the end, without stopping or branching – except at the end" [2:645]. An edge (B_1, B_2) means that block B_2 can follow directly after B_1 in the execution sequence [2:650], which means that the control flow from B_1 can pass directly to B_2 [41:13].

Fenton, Whitty, and Kaposi [29:146f.] define a *CFG* formally as a triple (G, a, z), where G is a finite directed graph and a and z are specific nodes of G. For this triple, the following must apply:

1. All the nodes apart from z have either fan-out 1 (these are called *procedural nodes*) or fan-out 2 (these are called *predicate nodes*). The node z has fan-out 0.
2. From the special node a (the *start node*), all the other nodes of G can be reached. The special node z (the *stop node*) can be reached from all the other nodes.

Program constructs which lead to many branches (for example `switch` in the language C) can be presented as a chain of predicate nodes. Alternatively, the definition and the theory based on it are slightly expanded, so that predicate nodes would have fan-out ≥ 2 [29:146].

2.2.3.1 Structuredness

According to Fenton, Whitty, and Kaposi [29:154ff.], *structured programs* are those which are only constructed from a certain number of basic control structures. Fenton, Whitty, and Kaposi [29:154] define a *composition operation*, by which a new CFG arises from two CFGs, F and G, by "replacing" in a defined way a procedure node x in F by G. This corresponds, for instance, to the replacement of a function call in the program by the definition of the function. The programming

method of *gradual refinement* [26:7] is based on a similar procedure. By repeatedly carrying out this composition operation, the class of *S-graphs* can be constructed out of a given set S of CFGs – the resulting CFGs are called *S-structured* [29:155f.]. Whitty, Fenton, and Kaposi [96] give an overview of the development of the term *structured programming*. In view of the widespread assumption that structured programs are easier to test, to debug, to understand and thus to modify, they emphasize that the verification of such assumptions requires recognized measures for software quality [96:55].

2.2.4 Issues: errors and desired additions/improvements

In an *issue tracking system*, the tasks which have to be carried out during software development are dealt with as so-called *issues*, which represent errors, desired improvements, desired additions or other tasks. Every issue has a type – ERROR, IMPROVEMENT, ADDITION, TASK or SUBTASK. For this study, ERRORS are treated as undesired, IMPROVEMENTS and ADDITIONS as desired and tasks as neutral.[41] Furthermore, an issue has other properties, in particular a unique identifier, the time at which it was created and at which it was entered in the issue tracking system, and the time at which it was solved and at which it was marked as completed.[42]

In the issue tracking system, an issue proceeds through various states (see fig. 2.7 on the following page):

- It is "opened", that is, entered into the system.
- It is assigned to an assignee and is then "in progress".
- It is marked as "resolved", because the processing has been completed. This can mean that a corresponding problem has been solved, but also that the issue was identified as the duplicate of another one.
- It is "closed", for instance, after another developer has checked the solution.
- It is "reopened", for instance, because a supposed solution has proven not to be one.

These states may be entered in a different order and may also be repeated.

By looking at issues, measures of external quality properties of the corresponding software product can be ascertained, for instance, the processing speed of issues (see section 3.2, p. 46). In section 4.2.1.3, p. 54 the analysis of an issue tracking system which was carried out for this study will be presented in more detail.

[41] These are the types used in EJABBERD. In other projects, different types than these may be defined.

[42] These are the properties used in EJABBERD with the issue tracking system JIRA. In other projects and with other issue tracking systems, different properties than these may be defined.

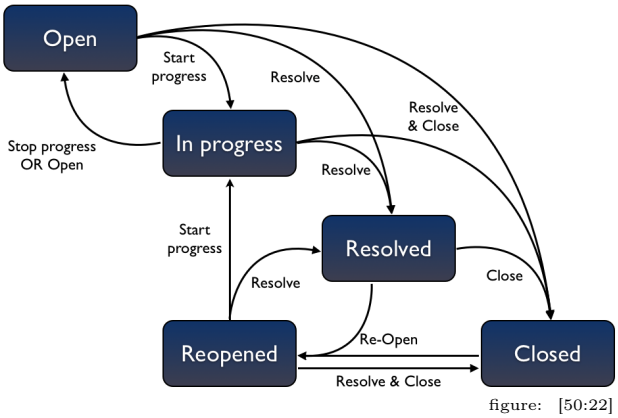


Figure 2.7 – States of an ·issue in the issue tracking system JIRA

2.3 Validation of software measures

> LOONQUAWL: "Forty-two! Is that all you've got to show for seven and a half million years' work?
> COMPUTER: I checked it very thoroughly, and that quite definitely is the answer. I think the problem, to be quite honest with you, is that you've never actually known what the question is."
>
> Douglas Adams [1:121]

Software measures are supposed to ascertain specific attributes of software quality. In order to establish whether measures do in fact accurately reflect the attributes to which they refer, various criteria and methods have been developed; these are collectively known as *validation*. ISO standard 25010 defines *validation* as "confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled" [49:20].

A distinction is made between internal validity, external validity and construct validity. *Internal validity* exists when the measure in question does in fact correctly measure the quality property which it is supposed to measure [97:407]. *External validity* means that the measure is connected in a specific way to a ***different***, external quality property [97:407f.]. *Construct validity* means that the definition and concrete implementation of a measure make it possible to correctly ascertain the property to be measured [65:16,25]. It follows from the reference to specific user requirements that the validation cannot be finally completed, but is a continuous process which has to be repeated for changed development processes, environments and projects [97:407,409f.].

Validation may be carried out *theoretically*, that is, by means of logical argument, and *empirically*, that is, by experimental testing [65:22] [65:17f.]. Meneely, Smith, and Williams [65:24f.] point to the fact that theoretical validation is often equated with *internal validation* and empirical validation with *external validation*, but that

in fact they deal with different aspects: “internal” or “external” denotes ***what*** is being validated, whereas “theoretical” or “empirical” states ***how*** the validation is carried out [65:24f.].

The question of the criteria according to which the validity of measures should be determined is still controversial [65:1]. Just like for the basic terms in software measurement, there is a multitude of competing concepts and terms (see section 2.1.2.2, p. 17): Meneely, Smith, and Williams [65:14ff.] have collected 47 different validation criteria in a *systematic literature review* [54].

In the following subsections, the validation criteria compiled by Meneely, Smith, and Williams [65] will be defined (numbering according to [65:14ff.]. These include in particular the criteria for external, empirical validity according to the IEEE standard 1061 [46:11f.], which are also included as a non-binding appendix in ISO/IEC standard 25020 [48:10f.]: ·association, ·discriminative power, ·suitability for predictions, ·rank consistency, ·replicability, ·trackability. Figure 2.8 on the following page shows the criteria selected for this study, with the criteria according to IEEE 1061 and ISO/IEC 25020 highlighted.

A number of criteria suggested in the literature were rejected for this study. These criteria and the reasons for excluding them are listed in appendix C.1, p. 113.

2.3.1 Internal validation

In the following sections, some theoretical and empirical criteria are defined with which to test whether a measure does correctly measure the property to be measured.

2.3.1.1 Theoretical internal validation

Appropriate domain, #3 The measure must be defined for all the instantiations of a property which actually occur and must not have any gaps in the domain [65:15].

Dimension consistency, #14 The ·measurement function of a ·derived measure must be a scientifically explained and well-understood connection [65:17]. This can, for instance, mean that different measures should not be connected in such a way that conclusions can no longer be drawn about the contribution of the individual components (see also section 2.1.1, p. 15).

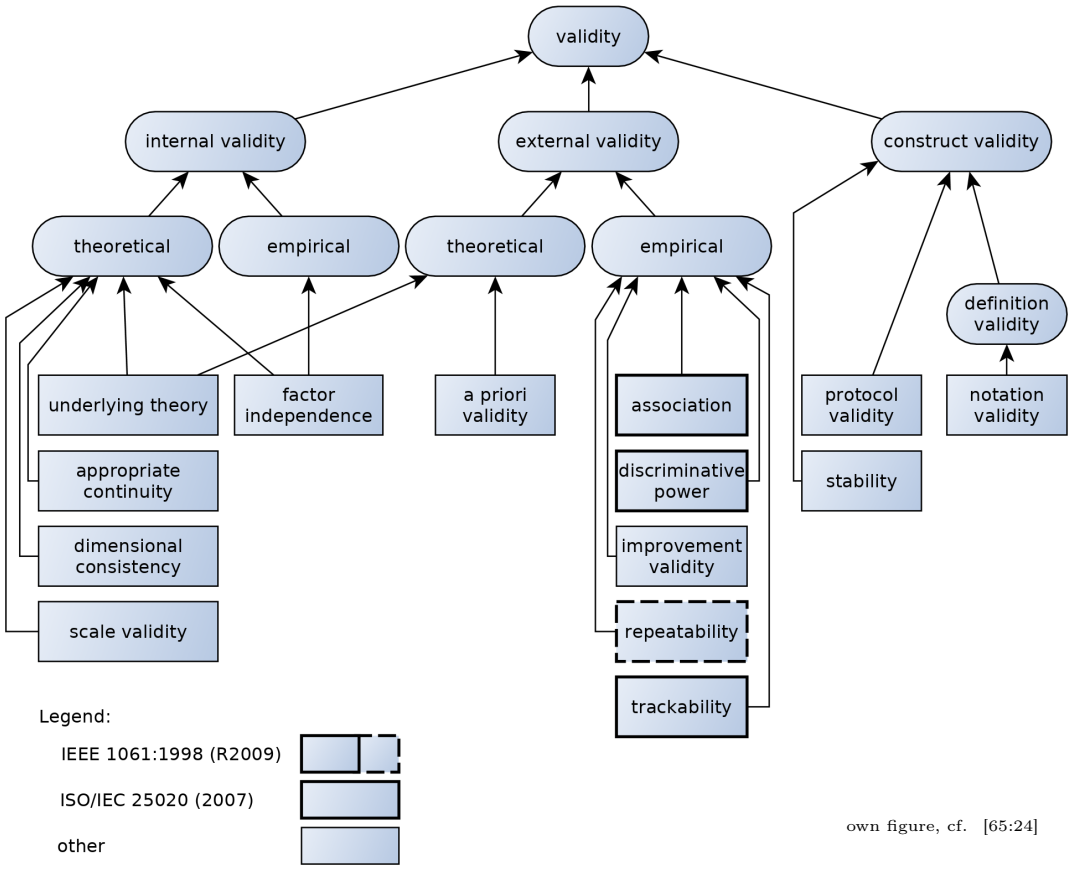


Figure 2.8 – Various validation criteria used in the literature, selected from Meneely, Smith, and Williams [65]

Factor independence, #18 The components of ·derived measures should be independent of each other [65:18]. This can be checked theoretically by looking for repeated occurrences of ·base measures. But empirically, the (undesired) correlation of the components should also be checked.

Scale validity, #40 The ·scale type of a measure should be indicated explicitly [65:21]. The validity of a specific operation on the measurements, such as a statistical calculation, depends on the scale type.

Underlying theory, #45 See section 2.3.2.1 on the next page.

2.3.1.2 Empirical internal validation

Factor independence, #18 See section 2.3.1.1.

2.3.2 External validation

Like internal validation, whether a measure is connected to an external property can be tested both theoretically and empirically. Below, some of the criteria used for this are given.

2.3.2.1 Theoretical external validation

Underlying theory and a-priori validity #45/#1 There should be a theoretical foundation for the construction of the measure, appropriate to the state of knowledge of the area of application (*underlying theory*) [65:22]. In particular, the presumed connection between attributes should be postulated ***before*** the testing, not simply in retrospect (*a priori validity*) [65:14]. This serves to avoid generalizing individual significant results gained by chance.

2.3.2.2 Empirical external validation

Association, #5 A measure should be immediately statistically correlated with an external quality property [65:15]. To be precise, there can only be a correlation with a ***measure*** for an external quality property.

Discriminative power, #13 A fixed threshold value should exist which distinguishes low-quality entities from high-quality ones [65:17]. Such a threshold value can aid in finding out, for instance, which parts of a software system must be most intensively reworked or tested.

Improvement validity, #19 A (new) measure should in some way present an improvement over previous measures [65:18], for instance, be more efficient or more precise. A common application of this criterion is the comparison with the number of lines as a "reference measure".

Replicability, #38 A measure should be empirically valid for various projects or stages of a project [65:21]. This requirement applies to all research results; it makes their robustness dependent on them being proven non-random.

Trackability, #43 A measure should change in the course of time parallel to an external quality property [65:22]. This is a weaker version of ·association, allowing for a temporal shift in corresponding characteristic values and measurements.

2.3.3 Construct validity

Definition validity, #12 The definition of the measure must be clear and unambiguous, in order to facilitate a precise, objective measurement [65:17]. Meneely, Smith, and Williams [65:17] require in addition that the measurement result differ from that of other measures. The latter requirement seems unnecessarily restrictive – for example, a program function can readily have just as many lines as parameters without this leading to confusion. One aspect is notation validity (#30), which applies when the measure is notated mathematically with precision and consistency [65:19].

Tool validity, #20 The measuring tool must measure correctly [65:18]. This criterion requires the verification of the measuring tool and/or a reference tool to check the measurement.

Protocol validity, #35 The measurement should follow a generally recognized "measurement protocol", that is, a ·measurement approach.

Stability, #41 A measure should give the same measurement values under the same conditions [65:21]. This includes independence from subjective judgements and presupposes ·definition validity.

2.3.4 Validation process

In the validation process, the internal and external validation criteria are checked theoretically and empirically, in order to establish whether a measure in fact constitutes a homomorphous representation of the empirical objects in the formal objects. Figure 2.9 on the next page illustrates this process with reference to the measuring of software ***products*** (in the figure, the sequence runs from bottom to top).

A real software product with internal and external properties (bottom centre in the figure) is initially modelled in an appropriate way for the purpose of measuring (see section 2.2, p. 23). Product measures only record the properties of these model entities, such as control flow graphs. Therefore, what type of useful statement about the software product can be derived from product measures also depends on the type and quality of the modelling of the product by the entities. Control flow graphs, for instance, abstract from the formatting of the program text, from comments, documentation and many more aspects which are important for the quality of the product. These limitations have to be taken into consideration when

applying product measures, in order to avoid deriving invalid statements about properties which are not modelled in the entity.

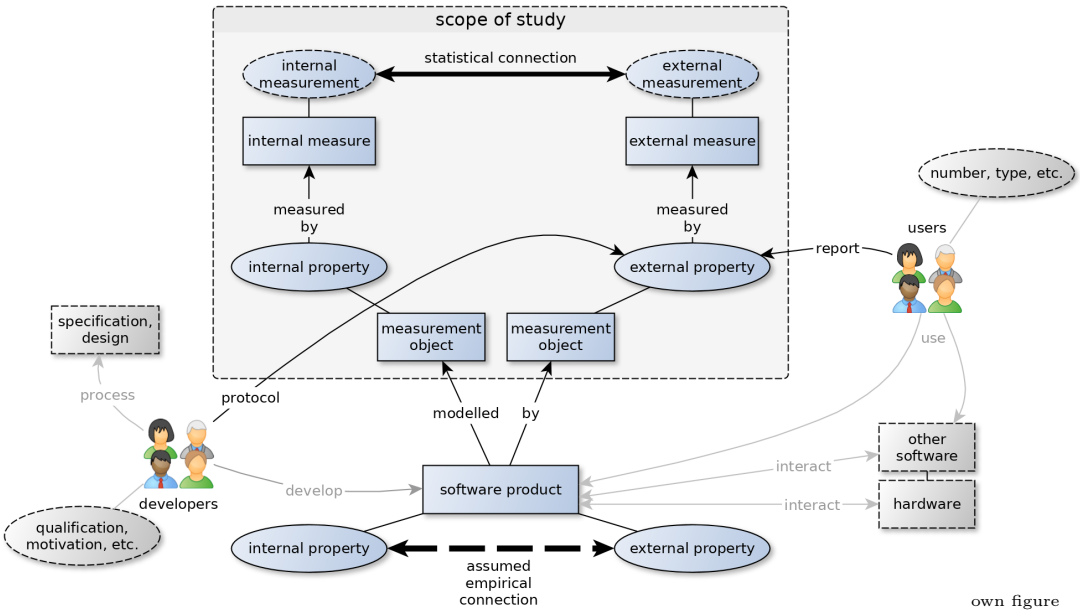


Figure 2.9 – Objects and stakeholders influencing the validation process

The type of measurement entity which is created depends on whether internal or external properties are to be measured: for example, a control flow graph might be used to measure internal properties such as the number of call relations between functions; a collection of ·issues from an issue tracking system, which represent errors or weak points in the software product, might be used to measure external properties such as processing efficiency. The properties of these entities are then measured – ***in place of*** the corresponding properties of the software product – by internal or external measures, giving as a result internal and external measurements (at the top of the figure). In the most widespread form of validation, external empirical validation of internal measures, a statistical analysis is then carried out to establish the correlations between these measures. Under the assumption that both the internal and external measures are internally valid, conclusions about the external validity of the internal measure can be drawn from the statistical correlation, that is, whether the internal measure is related to an external property.

Now further factors which affect software quality will be outlined, followed by a discussion of the challenges in investigating software errors as the basis for external properties.

2.3.4.1 Diverse factors influencing software quality

The quality of the software product is influenced by multiple factors; this is only indirectly reflected in software measures. In fig. 2.9 on the previous page, these influences are linked by arrows to the product and its models. They fall into three areas: influences relating to the software developers and to the users, and the influences of hardware and other software.

Software developers obviously exert a strong influence on software quality. Specialist qualification, motivation and performance during the development process are some of the factors which affect how this influence is exerted. Poorly-trained developers probably produce lower-quality software than well-trained ones; more highly-motivated developers possibly take more care and so produce somewhat less errors; developers who are under pressure through long working hours or other tasks may not achieve their full productivity in the product under investigation. Furthermore, developers rely on the preparatory work, such as specifications or draft documents, produced by other developers. Their quality also clearly influences the quality of the end product.

The working environment of the software product, which consists of hardware and other software, influences quality, as the software product must have certain properties to be able to interact with the environment.

The users also exert a certain influence on software quality, in that they experience the properties of a program in use and may urge the developers directly or indirectly to make modifications to the product or to future products. The next section will deal with this in detail.

All these factors operate as background variables on software quality. The aim of validation must therefore be to take the influence of these variables into consideration when evaluating software measures. This meets with significant difficulties because it is a rare occasion when the source code and comprehensive error reports for a long period are available [44:2] – concrete information about the developers, the users or the working environment are very difficult to obtain.

2.3.4.2 Empirical validation with reference to software errors

As a key issue in software development is the avoidance and, if necessary, prompt and efficient elimination of software errors, most of the empirical validation studies consider external properties which are related to errors in the software product.

Defective software in use can have damaging effects, requiring even greater effort to deal with the damage caused – whether it be a plane crash due to defective control software or data loss because of an error when saving a document. Software errors must be avoided or, if necessary, recognized as early as possible, because

the effort to rectify them increases exponentially in the course of the development process.

A distinction is made between cause of error, error condition and error effect [83:333]. The *cause of error* is a mistake or disturbance outside the software system. An *error condition* is the internal property of the software product which may lead to defective behaviour or failure of the software, the *error effect*. This property can exist permanently statically or develop dynamically when the program is running. Only static error conditions, called *errors* for short, will be considered here. According to Schlingloff [83:333], "software errors are always of a systematic nature and can be traced back to mistakes in the construction of the software." Various software measures such as ·cyclomatic complexity are assumed to be connected to the probability of such mistakes.

Large software systems probably always contain more ·errors than are recognized on the basis of observed ·error effects, so the *known errors* are only a fraction of all ·errors. In turn, only a fraction of the known errors is represented in potential data sources for empirical investigations. ·Issue tracking systems (see section 2.2.4, p. 26) are one such data source.

The issues in an issue tracking system which are of the type ERROR (see section 2.2.4, p. 26) represent a sample of all known errors. This sample is defective and distorted to a certain unknown extent. For instance, Hooimeijer and Weimer [43:35] report that in the MOZILLA project, 140 ERROR issues were created between 2003 and 2006 which related to the same ·error in a progress indicator. These 140 ERRORS were only gradually recognized and marked as duplicates, so that at any one time, this individual error was clearly overrepresented. Determining the quality and relevance of the ERRORS and other issues is a complicated problem as such, which is, however, rarely dealt with in empirical studies in the field of software engineering [9:122].

In order to research the connections between internal software measures and errors, errors must be linked to the affected program components. According to Bird et al. [9:125], the standard method, which will also be used in the present study, is to search the entries in the modification log of version control systems such as CVS or GIT for identifiers of ERRORS and assign to a program component those ERRORS which are noted in their change logs [9:125]. Frequently, however, only a quarter to a half of all ERRORS are noted in this way. According to Bird et al. [9:129], it is very complicated to find out how this sample of errors is distorted – which would make it possible to draw conclusions about the ERRORS which are not linked.

As errors can usually only be registered as ERRORS if they have caused an ·error effect, the distribution of the ERRORS among the program components depends not only on the assignation of actual ·errors, but also on the duration and frequency of use, as well as on the size of the components and the reporting behaviour of

the developers and users. As the life-span of a component or its duration of use increases, there is a greater probability that existing errors will occur and be reported. As a component increases in size, by contrast, it may be assumed that the probability of any individual error occurring will decrease, as the probability of the corresponding program part being run will tend to decrease [57:247]. A component which is full of ·errors but is rarely run can thus come off completely free of ERRORS. The frequency distribution of duration and intensity of use and thus of their influence on the number of known errors is largely unknown [44:12]. The method for assigning ERRORS and other ·issues to program components which was developed for the present study will be described in detail in section 4.2.3, p. 57.

2.4 Functional programming

> "The key issue for functional languages is to enable some of the elegance, clarity and precision of mathematics to flow into the world of programming."[43]
>
> Peter Pepper [76:2]

For the development of software, there are various fundamental approaches, which are known as styles, paradigms or models [77:892ff.]. One distinction made is that between ·imperative programming and ·descriptive programming [12:6] (see fig. 2.10, p. 37).[44] Imperative programming includes the predominant variants *procedural programming* and *object-oriented programming*, which include well-known languages such as PASCAL and C or C++ and JAVA. In this approach, programs consist of a series of instructions which are carried out step-by-step by the computer, calculating a result by explicitly modifying the state of the computer. That is, programs describe in detail ***how*** the result is to be calculated.

In contrast to this, in *descriptive programming* the focus is on describing ***what*** is to be produced; the description tends to abstract from the ***way*** it is produced [63:524]. Descriptive programming includes the variants *logical programming* and *functional programming*, with PROLOG or LISP and HASKELL as well-known languages.

In *functional programming*, programs are defined by a set of functions, which orient closely to the mathematical notion of function, that is, a function is a right-unique mapping from a definition set into a value set [76:15].[45] Functions consist of terms, which may contain constants, variables and function applications [76:20f.]. As is usual in mathematics, the value of an expression depends exclusively on

[44] ·Descriptive programming is often also called *declarative programming*.

[45] Pepper [76:15] speaks erroneously of "left-unique" when he evidently means "right-unique" [92:2].

the values of the variables and constants applied, but not on the expressions which were or are evaluated otherwise. For given parameters, the evaluation of such a term always results in the same value, that is, the term represents this value. Thus, in a given context of variable and constant values, terms can be replaced by their values without further effects [10:3]. This is known as *referential transparency* [63:524]. In contrast to the terms in functional programming, the instructions in imperative (for example, procedural) programming can have effects in addition to the calculation of values, which means they do not always arrive at the same result, even with the same parameters. These *side effects* may change the values of variables which are used in the instruction, so that the result of the instruction is also changed.

In functional programming, functions are treated like normal data structures, that is, they can be created in the same way, passed as parameters, returned as results and saved in data structures [63:524]. Functions are described as "first-class citizens" because the possibilities for processing them are not restricted in comparison with data objects.[46] Functions which use functions as parameters or which return functions as values, are known as *higher-order functions* [63:524f.].

2.4.1 Functional programming languages

In connection with the question what constitutes ·functional programming as a paradigm, Pepper [76:2] states that: "Unfortunately, all attempts to give something like a formal-mathematical definition for the difference between functional and imperative have failed."[47] Correspondingly, it is sometimes disputed whether a certain language can be classified as functional programming. Further, an all-purpose language cannot manage with exclusively mathematical functions. In particular, the interaction with the outside world in the input and output of data cannot be represented purely by functions.[48] To the question ***"So have we reached the limits of our paradigm?"***, Pepper [76:243] responds: "Certainly if we cling like purists to a dogma of the kind: 'Functional programming means working with the mathematical notion of function.' But if we look at the thing more pragmatically, then it is not an insistence on mathematical principles which is paramount, but the hope of achieving greater elegance, clarity and accuracy through an orientation to mathematical concepts."[49]

[46] Here objects are meant in the general sense of "things", not in the special sense of object-oriented programming.

[47] „Leider sind aber bisher alle Versuche, so etwas wie eine formal-mathematische Definition für den Unterschied zwischen funktional und imperativ zu geben, gescheitert."

[48] See Pepper [76:243ff.] on various possible solutions.

[49] „Sind wir also an die Grenzen unseres Paradigmas gestoßen?" – „Sicherlich dann, wenn wir puristisch an einem Dogma der Bauart kleben: ‚Funktionales Programmieren heißt, mit dem mathematischen Funktionsbegriff arbeiten.' Wenn wir die Sache aber pragmatischer sehen, dann steht nicht mathematische Prinzipienreiterei im Vordergrund, sondern die Hoffnung, durch

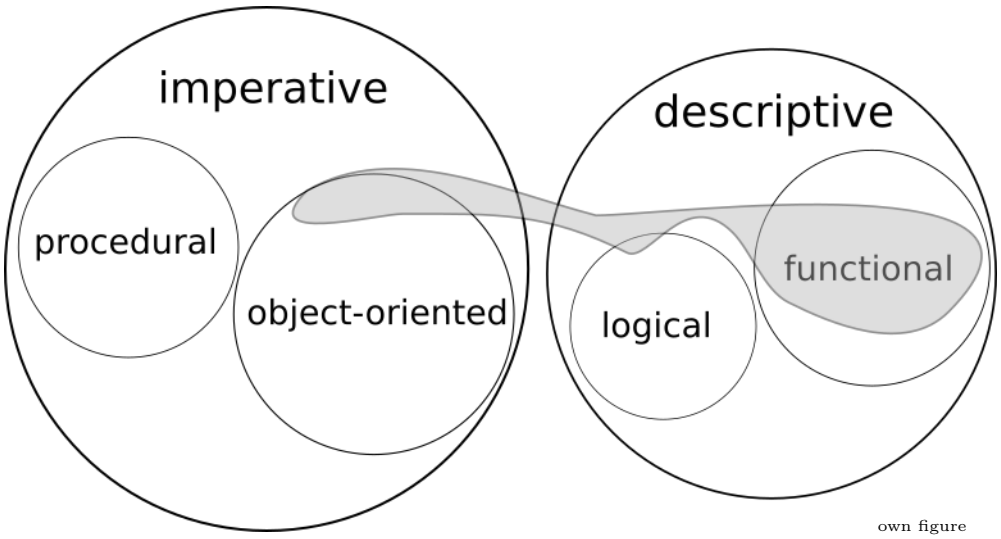


Figure 2.10 – Approximate classification of programming paradigms and languages. The circles depict programming paradigms, with the size approximating how widespread they are used. The grey area symbolizes a concrete programming languages.

In this study, *functional programming languages* are seen as those which ***support*** the functional approach to programming. Most concrete languages cannot be assigned to a single programming style, but unite elements from various approaches [12:7] [58:1] (an example is represented by the grey area in fig. 2.10). Further, some features or language constructs were adopted from the area of functional programming in languages whose origin is in imperative programming (for example, "anonymous functions" in C++11), and more recently some languages have been developed as mixed forms of several approaches, including the functional (for example, Ruby or C♯).

2.4.1.1 The language Erlang

The software system which is investigated in the present study is written in the functional, dynamically typed and strictly evaluated language Erlang [58:240f.]. Since 1987, the company Ericsson has been developing Erlang for applications in the field of telecommunications, which require an uncomplicated, efficient management of concurrent processes [58:240f.]. Erlang was influenced among other things by the logical programming language Prolog, which it resembles syntactically. The language corresponds to functional programming in the following respects:

eine Orientierung an mathematischen Konzepten größere Eleganz, Klarheit und Korrektheit zu gewinnen."

Listing 2.1 – An ERLANG module with functions `fact/1` and `doubler/0`

```
-module(mod_smpl).

fact(X) when X < 0 -> throw(error);
fact(0) -> 0;
fact(N) -> N * fact(N - 1).

doubler() -> fun(X) -> 2 * X end.
```

- In their validity scope, variables have either exactly one value or none at all [58:241]. This means that ·referential transparency exists for variables, which simplifies in particular the management of concurrency.
- Functions are "first-class objects", which means they can occur as variables, as parameters and as the result of functions.
- Every expression represents a value. (However, expressions do not always have the same value.)

An ERLANG program consists of *modules* in which *functions* are defined. Functions have to be *exported*, so that they are accessible from other modules. Functions of other modules can be *imported*, in order to be used like local ones. As in other languages, definitions used in common can be outsourced in so-called *header* files.

A function consists of one or more *clauses*. Listing 2.1 shows a module called `mod_smpl` with a function which consists of three clauses, and one which consists of a single clause. Clauses consist of head and tail; these are separated by `->`, in the example `fact(N)` and `N * fact(N - 1)`, for instance. The head contains the name of the function and the parameter list, the tail is an expression which defines the value of the clause. Across the system, functions are identified uniquely by their module, their name and their arity, using the following notation: `mod_bsp:fakul/1`. *Anonymous functions* can be created dynamically at run-time with `fun` expressions such as `fun(X) -> 2 * X end`, they can be assigned to variables or returned as values of functions. The function `doubler/0` in the example returns as the value a new function, whose value is double its argument.

Conditial expressions are defined by `case ... of ...` or `if ... then ...`, exception handling can be carried out with `try ... catch ...`. With `spawn(X)`, the function assigned to the variable `X` is started as a new process. If with `Pid = spawn(X)` the process number of this process is saved in the variable `Pid`, messages can be sent to this process with the *send operator* `!`: `Pid ! 42` would send the number 42 as a message. The process can wait for and react to specific messages by means of `receive`:

Listing 2.2 – Sending and receiving messages in ERLANG

```
self() ! hello
receive
        stop -> doStop();
        42 -> whatIs(42);
        X -> doSomething(X)
end
```

In the artificial example, the process first sends a message to itself (`self()`), then receives this message and reacts, depending on what it has received.

2.4.2 Special features with regard to software measures

Referential transparency means that there can be no loops with control variables in functional programming, as the control variable could not change its value. This means that instead of loop constructs, recursive functions are used – probably much more frequently than in imperative programming. Also, higher order functions can be defined, which use functions as parameters and apply them to other parameters, for example, lists.

In imperative programs, different instructions are sometimes carried out consecutively, gradually modifying the same data object, for example, when image data are subjected to different consecutive transformations. This is also impossible in functional programming, due to relational transparency, and it has to be resolved in a different way; for example, at every step a new data object is created, which is not modified but is only used to create a transformed data object.[50]

This and other differences in the available means of expression mean that functional and imperative programs are very different, even if they solve the same problem. That is why, even if software measures have been validated in relation to imperative programs, it is also necessary to validate them in relation to functional programs in order to draw conclusions from them.

[50] Here it is a question of the conceptual procedure. If a new ***logical*** object is created, the memory content affected does not necessarily actually have to be rewritten. Implementing the conceptual procedure efficiently is the task of the runtime system of the programming language.

3 Methods and tools

In chapter 1, p. 6, the present study was motivated and put in context; in chapter 2, p. 13, the basic concepts of software quality and software measurement, methods for modelling software products and procedures for validating software measures were presented and some special features of functional programming with regard to software measurement were discussed. Now the measures of internal and external quality features investigated will be defined and the analytic tool RefactorErl will be presented.

3.1 Selected measures of internal quality features

In order to ascertain internal quality features of software, a series of measures was selected, which will now be presented. As the focus of this study is not on the implementation of measures, but on their validation, the measures selected are in widespread use and can be readily ascertained with RefactorErl.

The ·base measures will be presented first, followed by ·derived measures which are based on them. Every measure has an Abbreviation, which will be used in the rest of this thesis. According to whether the entity of a measure is a whole software system, an individual module or an individual function, the abbreviation has the subscript "S", "M", "F"; if it can be applied to different components, "FM" or "FMS". The definitions are adapted so that modules and functions take the place of procedural and object-oriented component types: modules stand for classes and packages, functions for procedures and methods. Some hypotheses about connections to external quality features will be put forward in section 4.3.1, p. 63.

3.1.1 Base measures

3.1.1.1 Lines of code

The best-known and also simplest measure is the number of lines of program text. Lines can be subdivided into those which contain executable code and those which only contain comments.

Non-empty lines LOC_{FMS}[51] is the number of lines of a function, a module or a system which are not empty, that is, which contain executable code or comments. LOC_{M} of a module is the sum of the LOC_{F} of its functions plus other lines such as module or export declarations. LOC_{S} of a system is the sum of the LOC_{M} of its modules.

RefactorErl does not consider lines which only contain empty space as empty. It also does not count lines in headers.

Non-comment lines $\text{NCLOC}_{\text{FMS}}$[52] is the number of non-empty lines which are not comments [28:247].

Comment lines CLOC_{FMS}[53] is the number of comment lines [28:247].

3.1.1.2 Number of various program elements

Following LOC_{FM}, the simplest measures are those which merely state the number of specific program elements:

- NumMod_{S} is the number of modules in a system.
- $\text{Included}_{\text{M}}$ is the number of headers included in a module [79:57].
- $\text{Imported}_{\text{M}}$ is the number of modules from which functions were imported into a module [79:57].
- NumMac_{M} is the number of macros defined in a module [79:57].
- NumRec_{M} is the number of records defined in a module [79:57].
- NumFun_{M} is the number of functions in a module [79:57].[54]
- $\text{CallsIn}_{\text{M}}$ is the number of calls of internal functions of a module from external functions, that is, functions of other modules [79:58].
- $\text{CallsOut}_{\text{M}}$ is the number of calls of external functions from internal functions of a module [79:58].
- Calls_{M} is the number of calls of external and internal functions [79:57].
- $\text{NumClauses}_{\text{F}}$ is the number of clauses of a function [79:59].
- FanIn_{F} is the ·fan-in of a function [79:60] (see section 2.2.2, p. 25).
- FanOut_{F} is the ·fan-out of a function [79:60] (see section 2.2.2, p. 25).
- $\text{Returns}_{\text{F}}$ is the number of precursors of the ·stop node in the CFG of the function, that is, the number of expressions which potentially determine the value of the function [79:60].

[51] *Lines of Code*
[52] *Non-Comment Lines of Code*
[53] *Comment Lines of Code*
[54] Contrary to the statement in [79:57], version 0.9.12.01 of RefactorErl counts all functions which are called, even if their definition is not available.

- NumAnon$_F$ is the number of ·anonymous functions which are defined in the function [79:60]
- NumSend$_F$ is the number of ·send expressions in the function [79:60].
- RecBranch$_F$ is the number of points at which a function ***directly*** calls itself recursively [79:59].

3.1.1.3 Coupling relations

The term ·coupling refers to "dependence on and interaction between modules" [78:137]. In Erlang programs, coupling occurs when functions are imported and called between modules [vgl. 78:137].

Efferent function and module coupling OutFuns$_M$ is the number of functions in a module which call functions in other modules. Analogously, OutMods$_M$ is the number of modules whose functions are called from this module.[55]

Afferent function and module coupling InFuns$_M$ is the number of functions outside a module which call functions in this module. Correspondingly, InMods$_M$ is the number of modules from which functions are called in this module.[56]

3.1.1.4 Length of the longest non-recursive call path

MaxCall$_F$ is defined as the length of the longest non-recursive call path which emanates from the function [79:58].

3.1.1.5 Nesting depth of control structures

Nesting depth of `case` ·Case expressions (`case`) may be nested by giving the value of a branch as another case expression. MaxCase$_F$ is the maximum nesting level in one function [79:58f.].

Deepest nesting of `begin/end`, `case`, `fun`, `if` or `receive` Like case expressions, blocks, anonymous functions, binary branching expressions, receive expressions or exception handling structures may be nested. MaxNest$_F$ is the maximum nesting depth for all of these expressions [79:59].

[55] Compare ·efferent couplings in Martin [61:262f.].
[56] Compare ·afferent couplings in Martin [61:262f.].

3.1.1.6 Cyclomatic complexity

For a directed graph $G = (V, E)$ with p weakly connected components, the *cyclomatic number*

$$v(G) = |E| - |V| + p$$

indicates the number of linearly independent cycles [73:96].[57]

McCabe [62] supplements the ·control flow graph of a program with an edge from the end node to the start node. This makes the CFG strongly connected and the cyclomatic number of this graph corresponds to the number of linearly independent paths through the CFG [62:318]. This number is also known as the *cyclomatic complexity* of the program [62:308] (here denoted as CYC_F).

The cyclomatic number also indicates how many test cases are needed to achieve a complete branch coverage [57:239].[58]

3.1.1.7 Recursivity

As ERLANG like HASKELL does not contain any loop construct, presumably recursive functions will more frequently be defined for iterations [cf. 80:135]. Recursion exists when there is a cycle in the call graph of a function. If it is only the function itself which lies on this cycle, Ryder [80:135] speaks of *trivial recursion*; if other functions are called first on the cycle, he speaks of *non-trivial recursion*.[59]

- ISREC_F indicates whether a function is recursive [80:138] [79:61].
- TRIVREC_F indicates whether a function is ·trivially recursive [80:135]
- $\text{NONTRIVREC}_\text{F}$ indicates whether a function is ·non-trivially recursive [80:135]

3.1.2 Derived measures

3.1.2.1 Weighted functions per module

The weight[60] of every function of a module is determined by a defined measure μ (for example, cyclomatic complexity, see section 3.1.1.6). $\text{WMC}(\mu)_\text{M}$[61] is the sum of the μ-weights of the functions of the module.

[57] The ·cyclomatic number is also known as *cycle rank* [38:146] or occasionally as *Betti number* [37:626].

[58] Balzert [5:482] erroneously calls this the "minimum number of test cases" – but this obviously depends on the selected coverage measure. Instruction coverage is also possible with fewer than $v(G)$ test cases.

[59] These terms are not to be confused with those of *primitive recursive functions* [85:109].

[60] Chidamber and Kemerer [16] speak of "complexity".

[61] Compare *Weighted Methods per Class* in Chidamber and Kemerer [16:482].

3.1.2.2 Response for a module

Let the *response set* be defined as the set of all functions which are defined or directly called in a module [cf. 16:487].[62] RFC_M[63] is the cardinality of the response set of a module: $\text{RFC}_\text{M} = \text{NumFun}_\text{M} + \text{CallsOut}_\text{M}$.

3.1.2.3 Module coupling

CBO_M[64] is defined as the total number of modules with which a module is coupled [16:486], i.e. the sum of OutMods_M and InMods_M minus the number of modules to which both afferent and efferent couplings exist.

3.1.2.4 Instability

A module which depends on other modules must be modified if their external interfaces or functionality change. The more a module depends on other modules, the more frequently it must probably be modified. If many modules depend on one module, this means a pressure to modify this module as little as possible in order to avoid modifications in the dependent modules. A module that does not depend on any other module, but on which many depend, is therefore known as *stable*. A module which depends on many modules, but on which no modules depend, is called *unstable*.

These considerations lead to the definition of the instability measure Inst_M as the proportion of efferent couplings to all the couplings of a module (adapted from Martin [61:262]). Two variants, for function and module couplings, can be distinguished:

$$\begin{aligned} \text{InstF}_\text{M} &= \frac{\text{OutFuns}_\text{M}}{\text{OutFuns}_\text{M} + \text{InFuns}_\text{M}} \\ \text{InstM}_\text{M} &= \frac{\text{OutMods}_\text{M}}{\text{OutMods}_\text{M} + \text{InMods}_\text{M}} \end{aligned}$$

The instability measure takes on values from 0, “very stable”, to 1 “very unstable”.

[62] For reasons of efficiency, the transitive closure of the call relation is expressly not formed.
[63] Corresponding to *Response For a Class* in Chidamber and Kemerer [16:487].
[64] *Coupling between object classes*

3.1.2.5 Operation structuring

The average number of lines per function,

$$\textsc{AvgLOC}_\text{M} = \frac{\Sigma_\text{F}\text{LOC}_\text{F}}{\textsc{NumFun}_\text{M}}$$

is supposed to measure to what extent the program text is distributed sufficiently across functions. Very high values probably indicate functions which are hard to handle [60:28].

3.1.2.6 Operation complexity

The average cyclomatic complexity per line,

$$\textsc{AvgCyc}_\text{F} = \frac{\textsc{Cyc}_\text{F}}{\text{LOC}_\text{F}}$$

is supposed to measure how much branching in functions is to be expected [following 60:28].

3.1.2.7 Coupling intensity

The average number of different function calls per function,

$$\textsc{AvgCalls}_\text{M} = \frac{\textsc{CallsIn}_\text{M} + \textsc{CallsOut}_\text{M}}{\textsc{NumFun}_\text{M}}$$

is supposed to measure how strongly the functions interact with each other, that is, are coupled. Very high values could point to the fact that functions do not interact with the right "partners" [60:29].

3.1.2.8 Coupling distribution

The average number of modules called per function call,

$$\textsc{AvgOut}_\text{M} = \frac{\textsc{CallsOut}_\text{M}}{\textsc{CallsIn}_\text{M} + \textsc{CallsOut}_\text{M}}$$

is supposed to measure to what extent many modules are involved in the coupling [following 60:29].

3.2 Selected measures of external quality features

To measure external quality features of the software, a series of measures were selected, which will now be presented. The ·base measures will be presented first, followed by the ·derived measures which are based on them. Every measure has an ABBREVIATION, which will be used in the rest of this study. According to whether the measurement entity is a whole software system, an individual module, an individual function or an individual ·issue, the abbreviation has the subscript "S", "M", "F" or "I"; if it can be applied to different components, "FM" or "FMS".

3.2.1 Base measures

3.2.1.1 Number of issue types

NUMISS(T,V)$_{FMS}$ is the number of all ·issues of a ·type T which affect a function, a module or a system in a version V [following 36:654]. The types of issues are abbreviated as follows: "B" for BUGS, "N" for NEW FEATURES, "T" for TASKS and "I" for IMPROVEMENTS[65]

STARTISS(T,V)$_{FMS}$ is the number of all ·issues of a ·type T which affect a function, a module or a system in a version V ***for the first time***, that is, which were opened for this version [following 59:22].

ENDISS(T,V)$_{FMS}$ is the number of all ·issues of a ·type T which affect a function, a module or a system in a version V ***for the last time***, that is, which were closed during this version.

3.2.1.2 Processing time for issues

ITT$_I$[66] is the time-span between the opening and the closing of an issue, an indicator of the effort needed for a solution to the underlying problem [following 59:22,45].

[65] The context will make it clear that no "issue" is meant.
[66] "Issue Throughput Time" [59:22,45]

3.2.2 Derived measures

3.2.2.1 Error quota in issues

BugRate(V)$_{\text{FMS}}$ is the quota of ·errors among all issues which affect a function, a module or a system in a version. If there are no errors or no issues at all, BugRate$_{\text{FMS}}$ is zero.

3.2.2.2 New issues per month

StartRate(T,V)$_{\text{FMS}}$ is the number of new cases of type T in version V per month [cf. 59:22]:

$$\text{StartRate(T,V)}_{\text{FMS}} = \frac{\text{StartIss(T,V)}_{\text{FMS}}}{\text{Version lifetime in months}}$$

It should be noted that this number depends both on the error content or need to modify the software and on the intensity of use and other factors.

3.2.2.3 Project productivity

EndRate(T,V)$_{\text{FMS}}$ is the number of solved issues of type T in version V per month [following 8:39]:

$$\text{EndRate(T,V)}_{\text{FMS}} = \frac{\text{EndIss(T,V)}_{\text{FMS}}}{\text{Version lifetime in months}}$$

If no cases have been solved, the measure is zero. If there are no cases to be solved, the measure is not defined.

3.2.2.4 Mean time for processing an issue

The mean time for processing an issue MedITT(T,V)$_{\text{FMS}}$ is the median of the processing times for all issues of type T, which affect a specific function, a module or a system and which were solved within the lifespan of a version V. The length of time is given in hours.

3.2.2.5 Improvement rate

ImproveRate(V)$_{\text{FMS}}$ is the quota of Improvements among solved issues in version V [following 8:33]. If no Improvement issues were solved, the improvement rate is zero. If there are no issues to be solved, the measure is not defined.

3.2.2.6 Backlog management index – BMI

$\mathrm{BMI(V)_{FMS}}$ is the ratio of solved issues to issues created for version V per month [52:106]:[67]

$$\mathrm{BMI(V)_{FMS}} = \frac{\textsc{EndIss}(*,\mathrm{V})_{\mathrm{FMS}}}{\max(\textsc{StartIss}(*,\mathrm{V})_{\mathrm{FMS}}, 1)}$$

A BMI value below one means that more open issues are piling up than issues are closed; with a value above one, the number of open issues is being reduced.

3.3 RefactorErl as a tool for queries over program graphs

At present, there is only one tool – RefactorErl – which supports the acquisition of software measures for the language Erlang in a comprehensive and convenient way. Other tools only complete partial tasks, such as the creation of the ·syntax tree (Syntax Tools[68]) or the ·call graph (xref[69]) and require the implementation of further algorithms in order to query the required information and, for example, to create the ·control flow graph. The creation of ·control flow graphs is in general much more difficult for dynamically typed and/or functional programming languages than for statically typed or imperative programming languages [67:1] [88:2]. That is why the tool RefactorErl was selected for this study, in spite of the difficulties to be expected from a prototype.

RefactorErl[70] serves above all to refactor Erlang programs. On the basis of the static analysis required for this, it also makes it possible to query software measures and other information. It has been developed since 2006 in the Department of Programming Languages and Compilers of the Faculty for Computer Sciences at the Eötvös Loránd University in Budapest, Hungary. For the present study, the prototype in version 0.9.12.01 of 17 January 2012 was used. At the time that this study was completed, the current version was 0.9.12.14 of 20 April 2012, whose modifications are irrelevant for this research.

RefactorErl first creates an abstract syntax tree from the program code to be processed. In various analysis steps, this tree is extended by additional nodes and edges into a so-called *program graph*, which holds the results of the static semantic analysis: the call graph, statically analysable properties of dynamic constructs (for example, possible data types) and information about the data flow [53:269].

[67] As $\textsc{StartIss}(*,\mathrm{V})_{\mathrm{FMS}}$ can be zero, a denominator of at least one is set here.

[68] `http://www.erlang.org/doc/apps/syntax_tools/`

[69] `http://www.erlang.org/doc/man/xref.html`

[70] `http://plc.inf.elte.hu/erlang/`

3.3.1 Query language

To query the information stored in the program graph, a query language is used, which abstracts from the internal structure of the graph and offers a logical perspective corresponding to the structure of ERLANG programs: modules and headers, which define functions, macros and records, which in turn are made up of expressions. Figure 3.1 on the following page presents this logical perspective.

Similar to XPATH queries, queries describe sequences of steps in program graphs, where the steps are separated from each other with a dot: `mods` (top left in fig. 3.1 on the next page) selects all the module nodes, `mods.funs` all the function nodes, `mods.funs.vars` all the variable nodes, and so on. At every step, the selected set of nodes can be qualified by a predicate; for example, `mods[name = mod_smpl].funs[name = fact]` selects the function from listing 2.1, p. 38. Nodes have different properties (like `name` in the example), which can be queried as a final step or used in predicates (see fig. 3.1 on the next page). A detailed explanation of the query language is given in [79].

REFACTORERL provides some measures directly, in that they can be queried as properties of modules or functions. Unfortunately, the query language is too limited for the definition of some measures. For instance, it does not offer the possibility of counting arbitrary nodes or using variables in queries. However, query results can be assigned to ERLANG variables and then further processed. For this research, therefore, a measuring environment was developed which allows more complicated queries on the basis of a relational database system (see appendix B.2, p. 109).

3.3.2 Tool validity of RefactorErl

In its role as a ·refactoring tool, REFACTORERL is being tested both by the development team themselves and by other authors. The former include Tejfel et al. [87], who subject the tool to specification-based testing, and Bozó et al. [13], who deal with the testing of program transformations. Other authors include Deckers [20] and Jumpertz [51], who present successful approaches to the verification of REFACTORERL.

At present, no program errors are publicly known. Although the measuring functionality has not yet been explicitly researched, confidence in it is also strengthened by this, as it is based on the same program graphs as those used in refactoring. The measuring functionality will be tested in section 4.2.4, p. 60.

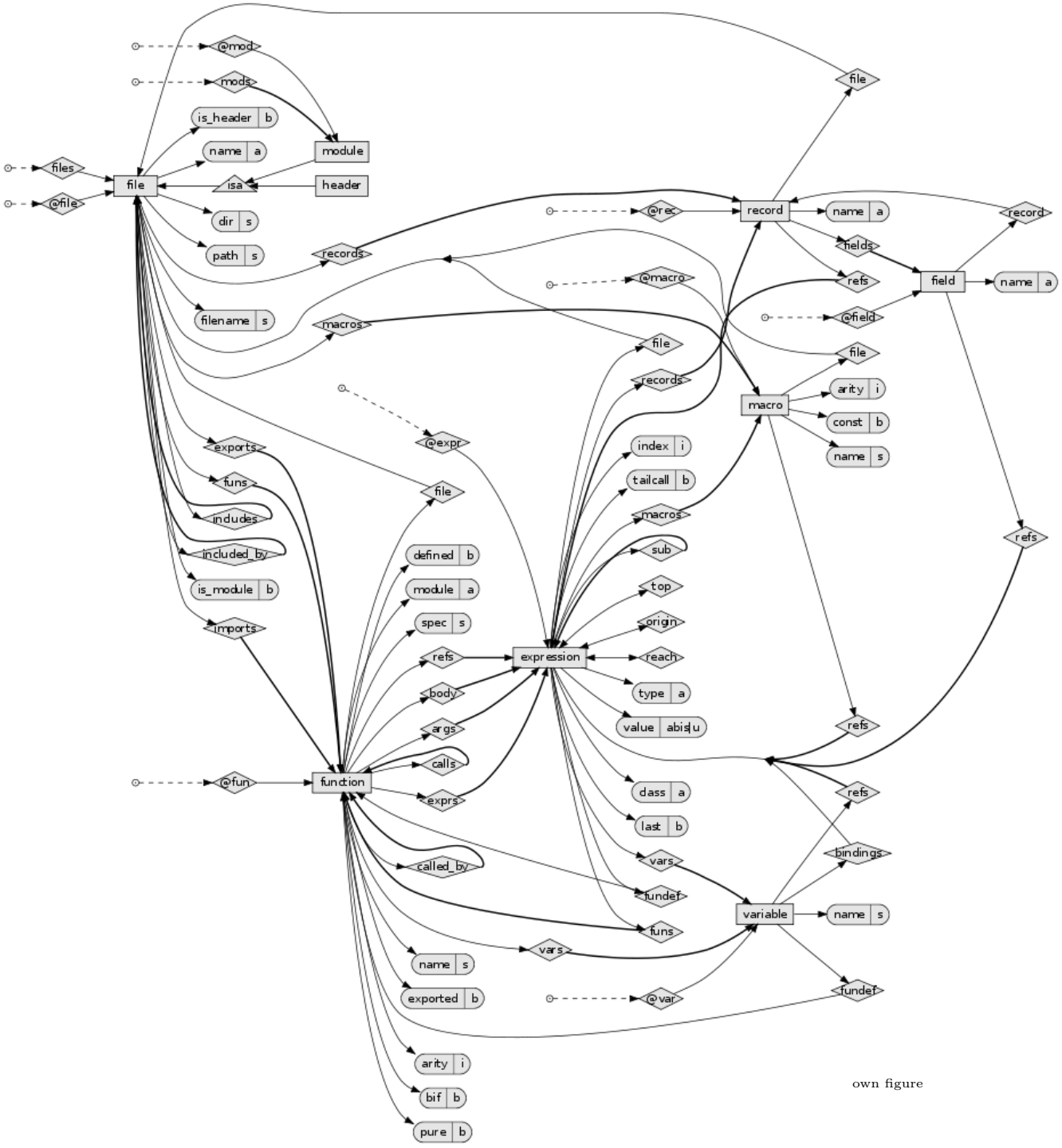


Figure 3.1 – Relations and attributes that can be queried with RefactorErl. The illustration is inspired by the Entity Relationship Model. Attributes are labeled with their data types: **a**tom, **b**ool, **i**nt, **s**tring, **u**nknown. Thin edges denote 1:1 relationships, bold edges denote 1:N relationships.

4 Empirical research

In the previous chapter, the measures to be validated and the tools to ascertain them were dealt with. Now the software product which was examined will be presented, as well as the structure of the research and the nature of the data obtained. Finally, some hypotheses regarding the considered measures will be formulated, the data statistically analysed and, on this basis, statements about the hypotheses will be made.

4.1 Research object ejabberd

For the present study, the research object has to have the following properties:

- The program text and information about known errors must be available.
- It must be mainly written in ERLANG.
- It should be as large and as frequently used as possible, in order to achieve reliable results.

Some large databases with completed measuring results such as FLOSSMetrics[71] or PROMISE[72] do not contain any ERLANG projects (and hardly any projects at all which use functional programming languages). With the exception of the runtime environment and standard library ERLANG/OTP itself, the available ERLANG projects are much smaller than the software systems which are considered in comparable research into imperative programming languages: a few ten thousand lines compared with several million or more lines.

EJABBERD is one of the most popular [68:432] server programs for the EXTENSIBLE MESSAGING AND PRESENCE PROTOCOL, XMPP [82:3] and known for its scalability and clustering capacity. With some 70 000 lines, it is one of the most comprehensive projects developed in ERLANG. The program text is available for all 29 release versions of EJABBERD. In addition, the version management system GIT also contains all approximately 2 600 revisions with information on the individual code modifications (*commits*) for the development period of over nine years. Error records are available in the *issue tracking system* JIRA. By the end

[71] http://flossmetrics.org/sections/deliverables/WP1
[72] http://promisedata.org/

of 2010, there were about 1 500 entries in this system, documenting errors or other reasons for modifications.

EJABBERD is also being used successfully in a communication system for tablet computers, which has been developed since 2010 in the Berlin company ESYS GmbH[73] which specializes in mobile measuring technology and PC network technology. The author implemented a JAVA library to use the XMPP protocol for this system, enabling the exchange of textual and graphic messages on the Android platform.[74] Intensive practical work with EJABBERD was the starting point for this research.

4.2 Structure of the study

The empirical study consists of four main phases: collection of the raw data, analysis and transformation of the data, gathering of the measurement values, statistical analysis of the measurement values. These phases will now be presented. Figure 4.1 on the following page illustrates the process progressively from the top to the bottom.

4.2.1 Data collection

The data for the research come from three sources (presented as cylinders in fig. 4.1 on the next page): the HTTP server with the release versions of ejabberd[75], the GIT version management system[76] and the JIRA issue tracking system[77].

4.2.1.1 Release versions

For each of the 29 published versions (releases) of EJABBERD (see table 4.1, p. 54), the program text is available to be downloaded as a package. The main part written in ERLANG is in module and header files, plus a few modules which are written in ·Abstract Syntax Notation One (ASN.1) and first have to be translated into ERLANG code with the ERLANG compiler. These ASN.1 modules and all other files (including scripts and documentation) are ignored. The functions and modules of various versions are abbreviated below as "function versions" or "module versions".

[73] `http://www.esys.de`, "Profil", accessed 26 April 2012.

[74] The JAVA library and its documentation can be found in the electronic appendix to this study.

[75] `http://www.process-one.net/en/ejabberd/archive/`, `http://www.process-one.net/en/ejabberd/downloads`

[76] `git://git.process-one.net/ejabberd/mainline.git`, `https://git.process-one.net/ejabberd/mainline`

[77] `https://support.process-one.net/browse/EJAB`

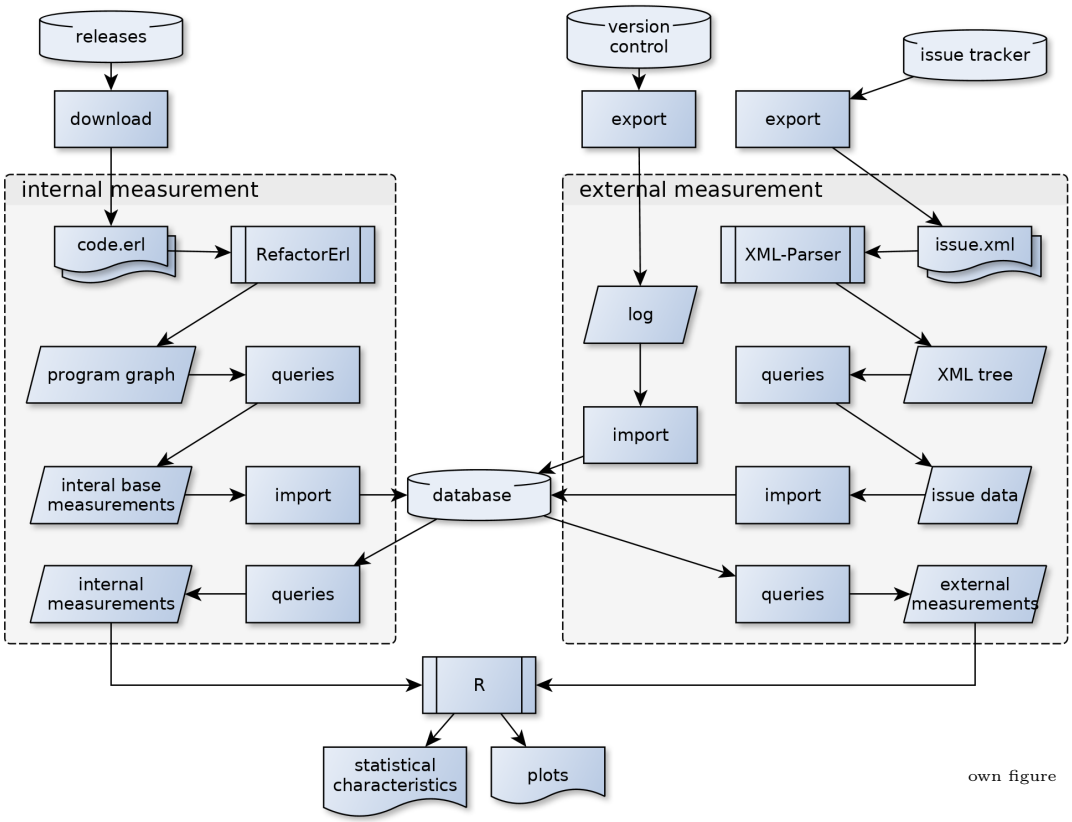


Figure 4.1 – Flow diagram of the empirical study

4.2.1.2 Modification records

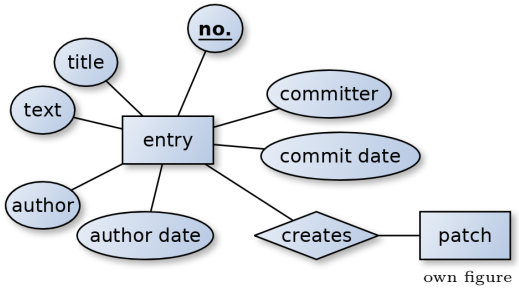
The version tracking system GIT keeps a record of modifications in which for every ·commit, various pieces of information can be retrieved, in particular the time the modification was made, a brief verbal description and the individual modifications to all affected files.

With GIT, the information about which modifications were made with the commit to which files can be retrieved in the form of PATCH files. In these files, the numbers of the modified lines in the original and the altered version of the code are given, as well as the lines themselves, with the information whether they were deleted, added or modified. For the present empirical research, the names of the affected modules are extracted from the file names and the affected functions are determined from the line numbers (see listing B.6, p. 111).

The verbal descriptions of the commits sometimes reference the unique identifier of the ·issue or ·issues which are being processed by the modifications of the commit. Via these references, some of the connections between issues and modules or functions can be determined (see section 4.2.3, p. 57).

Table 4.1 – Versions of EJABBERD, 13 November 2003 to 24 December 2011

Version	Date	No.	Version	Date	No.	Version	Date	No.
0.5	03-11-13	r.01	1.1.3	07-02-02	r.11	2.1.2	10-01-18	r.21
0.7	04-07-13	r.02	1.1.4	07-09-03	r.12	2.1.3	10-03-12	r.22
0.7.5	04-10-10	r.03	2.0.0	08-02-21	r.13	2.1.4	10-06-04	r.23
0.9	05-04-18	r.04	2.0.1	08-05-20	r.14	2.1.5	10-08-03	r.24
0.9.1	05-05-23	r.05	2.0.2	08-08-28	r.15	2.1.6	10-12-13	r.25
0.9.8	05-08-01	r.06	2.0.3	09-01-15	r.16	2.1.7	11-06-01	r.26
1.0.0	05-12-14	r.07	2.0.4	09-03-13	r.17	2.1.8	11-06-03	r.27
1.1.0	06-04-24	r.08	2.0.5	09-04-03	r.18	2.1.9	11-10-03	r.28
1.1.1	06-04-28	r.09	2.1.0	09-11-13	r.19	2.1.10	11-12-24	r.29
1.1.2	06-09-27	r.10	2.1.1	09-12-17	r.20			



own figure

Figure 4.2 – ER model of a log entry in the version tracking system GIT

4.2.1.3 Issue tracking system Jira

The EJABBERD developers administer their tasks in the issue tracking system JIRA. To ascertain the external measures (see section 3.2, p. 46), information has to be extracted from the issue tracking system. This is retrievable in various forms, including as XML files.[78]

The basic structure of an issue was already presented in section 2.2.4, p. 26. However, the format of issues in JIRA can be configured flexibly, so that initially it must be established which data about an issue are available in the concrete JIRA configuration for EJABBERD. To this end, the XML representations of all the issues are downloaded and an ·XML schema definition (·XSD) [56:24] is generated with the tool TRANG[79]. This is manually transferred to an entity-relationship-model of an issue (see fig. 4.3 on the following page), which presents all the properties of an issue. From the entity relationship model, a corresponding relational schema for the database is developed, in which all output data and measurements are

[78] See, for example, `https://support.process-one.net/si/jira.issueviews:issue-xml/EJAB-1415/EJAB-1415.xml`.

[79] See appendix B.1, p. 109.

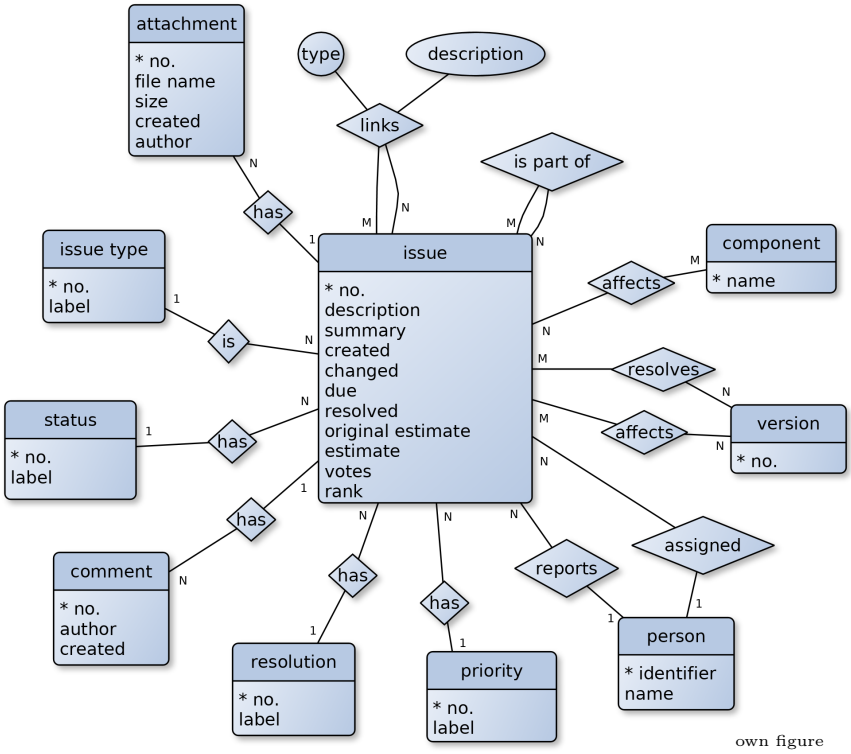


Figure 4.3 – ER model of an ·issue in the ·issue tracking system JIRA. (For a better overview, attributes of entities are depicted in UML style. Key attributes are marked with an asterisk (*).)

deposited.

The author knows of three tools which have been developed by others for extracting data from issue tracking systems: Luijten [59:8] reports in his study on issue tracking efficiency of the failed attempt to use the program ALITHEIA[80] to import issue data. As a result of this, he is developing his own system [59:14ff.]. Unfortunately, this is not publicly available nor can it be found at all. Furthermore, BICHO[81] promises to extract from issue tracking systems and make them analysable, but this proved to be still so error-prone that it required an unreasonable amount of effort, so the author also had to develop his own extraction program (see appendix B, p. 109).

As explained in section 2.3.4.2, p. 33, the issues in the issue tracking system represent a biased sample of all the existing problems or reasons for modifications, which moreover were identified ***after*** the release of a version of the software. ***Before*** release, many more, and different, problems will arise. In his research, Ryder [80:95] classifies all the code changes manually as corrections or improve-

[80] `http://sqo-oss.org`

[81] `https://projects.libresoft.es/projects/bicho`

ments; Bird et al. [9:129] gave some undergraduate students the task of manually classifying code changes in the APACHE project in order to acquire reliable error information. EJABBERD is too large for manual classification by one person and the second procedure is not available. For that reason, this study explicitly looks at ERRORS instead of errors, that is, at issues as the representation of a subset of the errors.

4.2.2 Data cleansing

Before the data are analysed, they have to be standardized in form and freed of defective elements.

4.2.2.1 Alignment of names

In ERLANG, the identifiers for modules and functions are ·atoms. As atoms may not begin with capital letters, the identifier of a capitalized module or function has to be framed in single quotes, which makes it an atom. Because such module identifiers no longer match the names of the files they are contained in, the single quotes will be removed in the rest of the investigation. This affects precisely the two modules which are produced by the ASN.1 notation, `'ELDAPv3'` and `'XmppAddr'`, which were excluded from the investigation anyway, as they are not developed as ERLANG code, but are only translated into ERLANG by the compiler.

4.2.2.2 Exclusion of incomplete or defective data

Before the analysis, incomplete or defective data are removed. The following criteria lead to exclusion of data:

1. ***Standard module:*** The associated module is part of the standard library, so the program text cannot be analysed. In any case, the respective contemporary version of the standard library would have to be analysed instead of the current one, which would go beyond the framework of this research.
2. ***Standard function:*** The associated function is not defined in the program text, but will be automatically produced during compilation. In this case, too, the program text cannot be analysed.
3. ***Code could not be loaded:*** The associated function definition could not be analysed by REFACTORERL. Reasons for this are syntax errors or the use of undefined macros. No measures that require an analysis of the program text are usable in this case.
4. ***Duplicates:*** A small quantity of data contains double measurements, which were determined by an early version of the measuring environment. These were removed in favor of the newer measurements.

Table 4.2 – Number of function versions excluded, with the reason for exclusion

Reason for exclusion	Number of function versions
Standard module	533
Standard function	1020
Loading error	2517
Duplicate	33

As table 4.2 shows, 4 103 function versions were excluded on the basis of the various exclusion criteria.

4.2.2.3 Exclusion of release no. 26

The 26th version of EJABBERD, Version 2.1.7, was only current for two days until it was replaced by Version 2.1.8, removing a serious error.[82] In spite of its short life-span, 30 issues were closed, and all the measures capturing the processing of issues over time would have been distorted. As it is otherwise identical to Version 2.1.8, it will be excluded from this investigation. The alternative of unifying its data with those from Version 2.1.8 was not adopted because of the anticipated conflicts between various measurements for the same components.

After completion of the data cleansing, the research covered 3 939 module versions at module level and 66 537 function versions at function level.

4.2.3 Mapping of issues to modules and functions

In order to be able to investigate the connections between internal and external measurements of software systems, it has to be determined which internal and external measurements affect the same releases, modules or functions. It is already known which program components the internal measurements belong to; for the issues in the issue tracking system, this has not yet been determined.[83] The assignation is carried out in two stages, which are explained below:

1. ***Direct*** connections between issues and program components are extracted from the modification record and the issue tracking system.
2. ***Indirect*** connections are derived from direct connections and the lifetime period of releases and issues.

[82] See `http://www.process-one.net/en/ejabberd/release_notes/release_note_ejabberd_2.1.8/`, last accessed 9 May 2012.

[83] By contrast, in the study of Graves et al. [36:655], this information was already available in the issue tracking system.

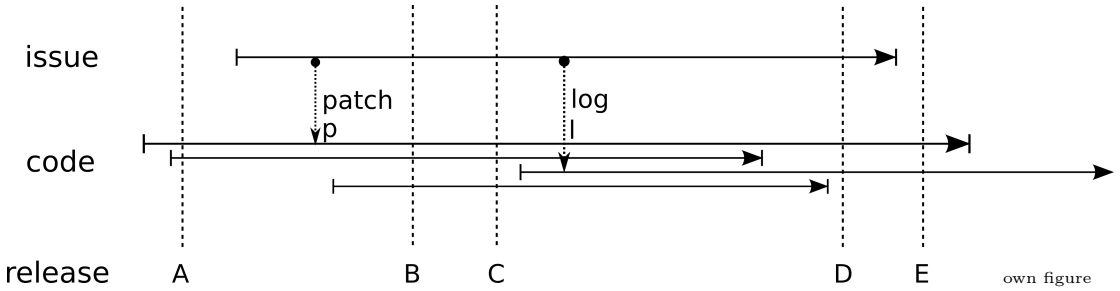


Figure 4.4 – Mapping of an issue to code parts via patches and the change log. The lifetime of an issue or code part (module or function) is depicted by an arrow from the time of creation up to the time of solution or deletion, respectively. In the example, via a patch and an entry in the change log it is known which code parts an issue affects at the times p and l. The issue affects this code part for all releases whose lifetime overlaps it's own lifetime, i.e. A, B, C and D.

4.2.3.1 Direct connections

The issue tracking system only indicates approximately for which versions and for which area (for instance, "multi-user chat" or "documentation") an issue is relevant; this barely makes it possible to connect them with concrete modules or functions. However, issues and program text can be connected with each other in two ways: firstly, through PATCH files which have been deposited in the issue tracking system and secondly through entries in the modification record referencing the unique number of an issue.

The publication of a patch at a specific point in time signals that the code sections affected by the patch are part of the problem that the patch is supposed to solve. This problem must have arisen at a point in time before the publication of the patch. A conservative assumption is that the problem has only existed since the last version before publication of the patch. The affected files and lines are retrieved from the PATCH file. Since in ERLANG, module and file names correspond, the modules are not organized hierarchically and the initial and final lines of every function are known, a direct link can be made in this way between an issue and a function (see fig. 4.4). To this end, the extraction program which has been developed downloads the PATCH files for each issue as a file attachment and extracts the module and function names. Via the date of the attachment, it identifies the EJABBERD version which was current at that time.

Finally, it enters in the central database a connection between the respective issue and the extracted modules and functions in the corresponding version (see fig. 4.4).

In the version control system GIT, a PATCH file can be produced corresponding to a modification entry; the affected modules and functions can likewise be extracted from this. For modifications which affect several issues, defective connections may be made, as in the PATCH file it is no longer possible to distinguish

which issue the modification belongs to. For this reason, Luijten [59:16], who is researching the connections between software measures and issue information for object-oriented programming languages, does not make detailed connections between issues and program text. In EJABBERD, 38 of the 2 641 modification entries mention more than one issue; a total of 84 out of 1 470 issues are affected by these mixed connections.[84] In order to avoid false positive assignations, the corresponding modification entries are ignored for the connections.

With this technique, a total of 726 out of the 1 470 issues, 49 per cent, are linked to program components. At the function level, all these issues are assigned to 4 374 function versions through 5 840 connections, which corresponds to 6,5 percent of all the versions. At the module level, the same set of issues is assigned to 1 448 module versions by 3 347 connections, corresponding to 36,7 percent.

In the next section, it will be shown how this result can still be improved.

4.2.3.2 Indirect connections

Assuming that issues are closed within the life-span of that EJABBERD version in whose lifetime the underlying problem has been solved, the final date of the issue indicates for which version of the affected functions and modules the connection can be extended backwards in time. In fig. 4.4 on the preceding page, for example, this is Version D for patch p, because it is the last version which was published before the final date of the issue connected to the patch. In this way, the connection of the issue, which was initially established via patch p with version A of the affected functions and modules, is extended to all versions up to and including D. Put more generally, the indirect connections of an issue are produced for all versions whose life-span overlaps with that of the issue.

With this procedure, which has not been found in the literature to date, the number of connected function versions triple to 14 571, while the number of connected modules increases 1.4 times to 2 015 module versions. Thus, through direct and indirect connections 21.8 percent of all functions and 51.2 percent of all modules were connected. The number of connected issues obviously remained the same. Table 4.3 on the next page shows the distribution of all ·issues and the connected issues among the different types (see section 2.2.4, p. 26), as well as the respective proportion of all connections, both at function level and at module level. The correspondence between the distribution of the connected issues and the distribution of all issues indicates that the sample of connected issues is not distorted. It is striking that NEW FEATURES and IMPROVEMENTS have disproportionately large numbers of connections, that is, they are connected to a disproportionately large number of program components. Further research could look into this observation more closely (see section 7.2, p. 93).

[84] The figures are valid for the period up to 17 March 2012.

Table 4.3 – Distribution of issues and mappings over the different types. (Figures in percent. Sums of more than 100 percent possible due to rounding.)

Type	Proportion of issues: all	mapped	Proportion of mappings
Error	52	49	22
New Feature	14	14	29
Task	7	3	4
Improvement	30	34	45

4.2.4 Tool validity of the measuring system

With a prototype like RefactorErl, errors are even more likely than with more fully developed tools. While RefactorErl is repeatedly tested as a refactoring tool (section 3.3, p. 48), there is no known research specifically testing its measuring functionality. For researching the ·tool validity, the whole measuring system consisting of RefactorErl and all the supporting programs and scripts is now considered as the tool. The standard programs and archives (appendix B.1, p. 109) are not examined in detail, because it can be assumed that they function correctly.

To test a sample of the measuring results, the following values are determined for all 29 release versions of ejabberd, independently of RefactorErl:

1. Number of modules
2. Number of functions
3. Number of non-empty lines

These values were selected for comparison because they can be determined easily, so that errors in the control tool can be ruled out. However, the values are obviously not independent of each other: in all three comparisons, modules which were incorrectly imported lead to discrepancies; incorrectly imported functions also lead to discrepancies in the number of lines.

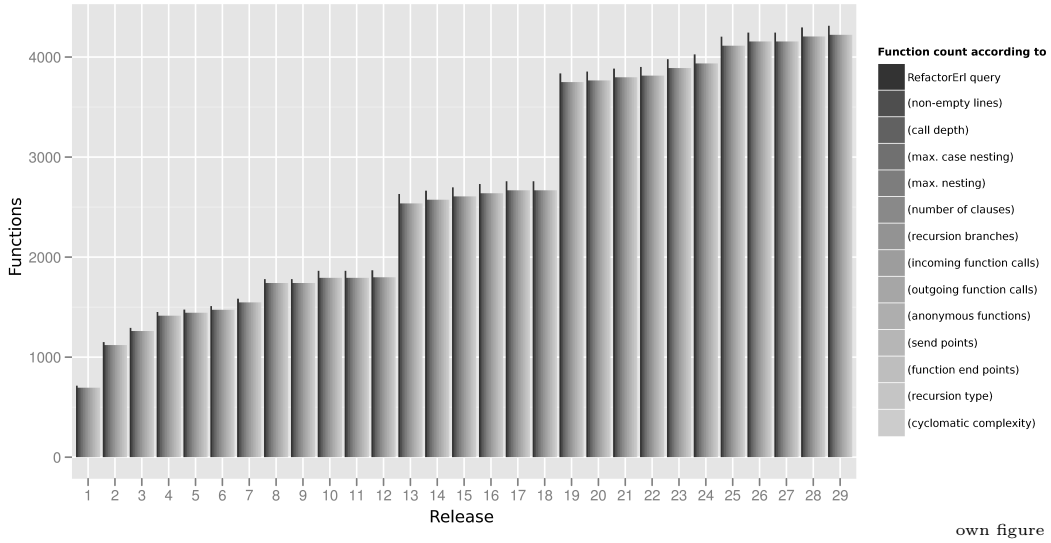
Discrepancies can arise because RefactorErl can only import code which can also be compiled in the operating environment. For example, if macros from missing archives are used, the import will fail so that no information or only limited information can be retrieved for the affected module.

4.2.4.1 Number of modules per version

For every release, the number of the modules imported by RefactorErl is compared with the number of Erlang files (see listing B.1, p. 110). This is a good approximation, as every module has to be in a separate file and every

Erlang file has to contain a module. The results of the two counts correspond, so that the first test of tool validity is successful.

4.2.4.2 Number of functions per version



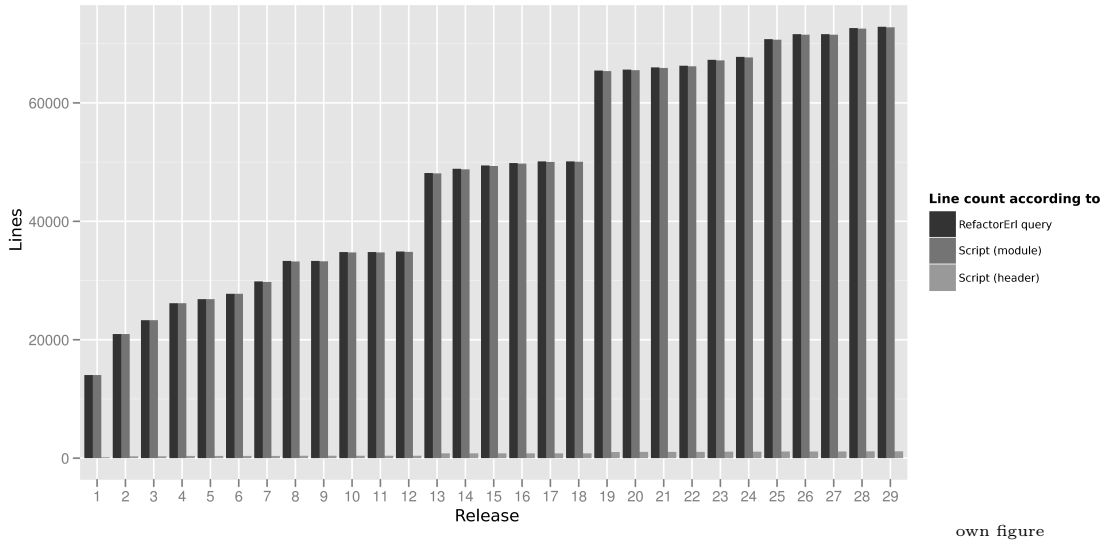
own figure

Figure 4.5 – Comparison of the number of functions in ejabberd. The number of functions was counted with RefactorErl on the one hand (black bars), and on the other hand, the functions were counted for which measurement values of RefactorErl's standard measures are available (grey bars, compare also listing B.3, p. 111).

For every release of RefactorErl, the number of functions is retrieved from the program graph and compared with the number of imported functions (listing B.3, p. 111). Only the functions of modules which were imported without error are counted. Figure 4.5 shows that the number of function measurements is identical for every measure. The measurements from RefactorErl are consistently slightly higher because it also counts undefined called functions.

4.2.4.3 Number of non-empty lines per version

For the number of lines (LOC) in every release, the figures for RefactorErl are compared with the results of listing B.5, p. 111. Determining the number of lines is not trivial for RefactorErl as, in order to count the lines, it first has to reconstruct the original program text from the program graph. This is therefore an acceptable test of the correctness of the program graph, from which all the basic measurements are retrieved.



own figure

Figure 4.6 – Comparison of the number of lines (LOC) in EJABBERD, determined via REFACTORERL and listing B.5, p. 111.

The results of listing B.5, p. 111 largely correspond with those of the measuring environment (see fig. 4.6). The differences amount to a negligible two lines in one single module from release no. 7; no explanation could be found for this, however.

4.3 Statistical examination

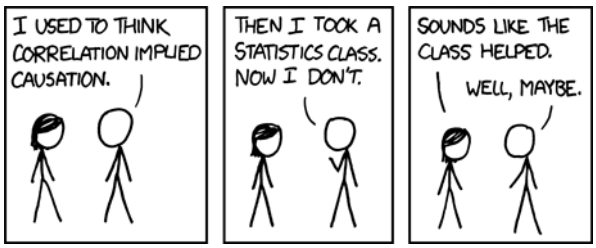


figure: [71:110120]

All the functions and modules from the remaining 28 versions[85] of EJABBERD will now be examined together. There will be no separate examination of the individual versions, to offset the unknown influence of the changing intensity of development work and use. This procedure corresponds to that of Hopkins and Hatton [44]. In contrast, Ryder [80] reduces the data for the whole life-span of the systems he studied to the maximum values of the internal measures per function and to the total number of corrections up to a fixed point in time (see section 1.2.0.2, p. 9). In this way, low measurement values and modification counts are underrepresented in his data set. By comparison, the approach selected here has the advantage of

[85] See section 4.2.2.3, p. 57.

taking all intermediate stages in the development of the program components into account. Depending on the measures under consideration, every module and every function is assigned its internal measurement, as well as its external measurement via the issues which are connected to it.

4.3.1 Hypotheses

The hypotheses examined correspond to the validation criteria ·association, ·discriminative power and ·trackability (see section 2.3.2.2, p. 30). They are either directly adopted from other authors (especially from Ryder [80] as the only known similar study with reference to functional programming) or are derived from informal programming guidelines (*Program Development Using Erlang - Programming Rules and Conventions* [26] as well as Cesarini and Thompson [15]). The hypotheses only affect the function or module level. Whole systems are not considered.

The ·selected level of significance is $\alpha = 0.05$. In view of the results in Berg [6] and Ryder [80] (see section 1.2, p. 7), correlation coefficients with a value below 0.2 are considered as negligible, those with a value between 0.2 and 0.5 as "weak" to "moderate" and all higher values as "strong".

Unless otherwise noted, the null hypothesis for the criterion ·association is that the values of the correlation coefficient are smaller than or equal to 0.2. The research hypotheses presented here with regard to the criterion ·association should be viewed as alternatives to this null hypothesis. If the hypothetical correlation is not denoted more precisely, its absolute value has a strength of > 0.2.

For the criterion ·trackability, in the first step it will be tested against the null hypothesis that the rank correlation coefficient equals zero. In the second step, the strength of the rank correlation coefficient will be rated according to the above rule. The criterion is met if a significant correlation with a strength of more than 0.2 is observed.

For the criterion ·discriminative power, proof of a significant correlation is sufficient; where appropriate, the strength of the correlation will also be ranked.

4.3.1.1 Basic assumptions

Informal statements about external quality properties in the consulted literature will be operationalized according to the following considerations, which cannot be tested in this study:

1. If the external property is not given more precisely, it will be assumed that ·maintainability or one of its aspects is meant. "Comprehensibility" will be seen as one of these aspects.

2. For the following external measures, a positive correlation with maintainability is assumed:
   - EndRate(T,V)$_{FM}$ – the average number of solved Errors per month.
   - ImproveRate(V)$_{FM}$ – the percentage of Improvements in the solved issues during the life-span of a version.
   - BMI(V)$_{FM}$ – the ·Backlog Management Index, that is, the ratio of solved issues to newly opened issues.
3. A negative correlation with maintainability is assumed for the following external measures:
   - NumIss(B)$_{FM}$ – the number of issues for a version of a function or a module.
   - BugRate$_{FM}$ – the share of Errors in all the issues of a version of a function or a module.
   - MedITT(B)$_{FM}$ – the median processing time of Errors in a version of a function or a module.

The sources given below refer to the statements from which the hypotheses were derived.

4.3.1.2 Hypotheses about association and trackability

To test the validation criteria ·association and ·trackability, the correlation and rank correlation between pairs of measurements will be investigated, where one of the measures is internal and one external.

Hypothesis AP1 Chidamber and Kemerer [16:482] hypothesize, in spirit, that WMC(Cyc)$_M$, the summed cyclomatic complexity of all functions of a module, correlates negatively[86] with ·maintainability. Therefore, the following sub-hypotheses are put forward here:

1. WMC(Cyc)$_M$ correlates positively with NumIss(B)$_M$.
2. WMC(Cyc)$_M$ correlates positively with BugRate$_M$.
3. WMC(Cyc)$_M$ correlates positively with MedITT(B)$_M$.
4. WMC(Cyc)$_M$ correlates negatively with BMI$_M$.
5. WMC(Cyc)$_M$ correlates negatively with EndRate(B)$_M$.
6. WMC(Cyc)$_M$ correlates negatively with ImproveRate$_M$.

Hypothesis AP2 For CBO$_M$, too, Chidamber and Kemerer [16:486] suspect a negative[87] correlation with the ·maintainability of software [cf. 5:474]. This results in the following sub-hypotheses, analogous to WMC(Cyc)$_M$:

[86] In the German original of this thesis, it erroneously says "positively" here.
[87] In the German original of this thesis, it erroneously says "positive" here.

1. CBO_M correlates positively with $\text{NumIss(B)}_\text{M}$.
2. CBO_M correlates positively with BugRate_M.
3. CBO_M correlates positively with $\text{MedITT(B)}_\text{M}$.
4. CBO_M correlates negatively with BMI_M.
5. CBO_M correlates negatively with $\text{EndRate(B)}_\text{M}$.
6. CBO_M correlates negatively with $\text{ImproveRate}_\text{M}$.

Hypothesis AP3 Chidamber and Kemerer [16:487] argue that RFC_M correlates negatively[88] with the ·testability and ·maintainability of software. The following sub-hypotheses will be examined here:

1. RFC_M correlates positively with $\text{NumIss(B)}_\text{M}$.
2. RFC_M correlates positively with BugRate_M.
3. RFC_M correlates negatively with BMI_M.
4. RFC_M correlates negatively with $\text{EndRate(B)}_\text{M}$.
5. RFC_M correlates negatively with $\text{ImproveRate}_\text{M}$.

Hypothesis AP4 With regard to Haskell, Ryder [80:148] observes moderate to strong correlations (between $r = 0.47$ and $r = 0.57$) between the fan-out of functions and the number of error corrections. Therefore, the following hypothesis will be tested against the null hypothesis that $r \leq 0.5$:

- FanOut_F has a strong positive correlation with $\text{NumIss(B)}_\text{F}$.

Hypothesis AP5 For ·instability, Spinellis [86:348] suggests a positive correlation with the frequency of program modifications, which is operationalized by the two variants of Inst_M as follows:

1. InstF_M correlates positively with $\text{StartRate}_\text{M}$.
2. InstM_M correlates positively with $\text{StartRate}_\text{M}$.

Hypothesis AP6 Hopkins and Hatton [44:8] report an unexpectedly negative correlation of the maximum nesting of `if` instructions and the number of errors. This leads to the following sub-hypotheses:

1. MaxCase_F correlates negatively with $\text{NumIss(B)}_\text{F}$.
2. MaxNest_F correlates negatively with $\text{NumIss(B)}_\text{F}$.

For these two research hypotheses, the null hypothesis is that there is no negative correlation, that is, $r \geq 0$.

[88] In the German original of this thesis, erroneously a positive correlation was implied here.

4.3.1.3 Hypotheses about discriminative power

Hypothesis T1 In *Program Development Using Erlang - Programming Rules and Conventions* it is recommended that `case`, `if` and `receive` should not be nested by more than two levels [26:23]; therefore, the following sub-hypotheses are formulated here:

1. Starting from a value ≥ 3 of $\text{MaxCase}_{\text{FM}}$, there are significantly more errors.[89]
2. Starting from a value ≥ 3 of $\text{MaxNest}_{\text{FM}}$, there are significantly more errors.

Hypothesis T2 On the basis of the recommendation in *Program Development Using Erlang - Programming Rules and Conventions* that the number of lines of modules should be limited to 400 [26:23], it is postulated:

- A module of more than 400 lines contains significantly more errors.

Hypothesis T3 Moores [69:48] calculates a threshold value for Prolog programs of 30 ± 5 rules with different names or arities, above which significantly more errors occur.[90] As Erlang and Prolog are syntactically similar and are both descriptive programming languages, the following hypotheses are transferred:

- From about 30 functions, a module contains significantly more errors.
- From about 40 functions, a module contains significantly more errors.

Hypothesis T4 From another recommendation in *Program Development Using Erlang - Programming Rules and Conventions* [26:23], the following hypothesis is derived:

- From about 15 to 20 lines, a function contains significantly more errors.

Hypothesis T5 For various measures, including AvgLOC_{M}, Marinescu and Lanza [60:28] speculate that $1.5 \cdot (\bar{x} + s)$ is a reasonable threshold level for the error-proneness of program components. As table D.2, p. 118 shows, the measure clearly does not have a normal distribution. For this reason, the threshold value is initially formed from the median and the mean deviation from the median, although for comparison the threshold level which is based on invalid assumptions is also investigated. The two variants of the hypothesis are:

[89] On the meaning of "significantly more errors", see section 4.3.5, p. 72.
[90] t-test, $p = 0.006$

1. For AvgLOC$_{\mathrm{M}}$ significantly more errors occur from the threshold value

$$1.5 \cdot (\tilde{x}_{0.5} + \tilde{d}_{0.5}).$$

2. For AvgLOC$_{\mathrm{M}}$ significantly more errors occur from the threshold value

$$1.5 \cdot (\bar{x} + s).$$

Hypothesis T6 Analogously to hypothesis T5, it is postulated [following 60:29]:

1. For AvgCalls$_{\mathrm{M}}$ significantly more errors occur from the threshold value $1.5 \cdot (\tilde{x}_{0.5} + \tilde{d}_{0.5})$.
2. For AvgCalls$_{\mathrm{M}}$ significantly more errors occur from the threshold value $1.5 \cdot (\bar{x} + s)$.

4.3.2 Notes on the figures

As there are sometimes different definitions of the types of figures used, the conventions adopted in this work will be briefly presented.

4.3.2.1 Box plot

The box reaches from the lower to the upper quartile ($\tilde{x}_{0.75}$ and $\tilde{x}_{0.25}$). Values from 1.5 times the quartile distance $d_Q = \tilde{x}_{0.75} - \tilde{x}_{0.25}$ [89:73] from the upper or lower margin of the box are seen as outliers. The "antennae" reach to the smallest and the largest non-outlier, respectively, and are ***not*** enclosed by horizontal lines. The median is presented as a horizontal line, the arithmetic mean as a star.

4.3.2.2 Scatter plot

The axes begin with zero, even if this is not marked. The points are presented semi-transparently, so that individual points are grey and several overlapping points look darker. The marginal distributions are presented by box plots parallel to the axes. Dashed straight lines parallel to the axis show the outlier margins of the box plot. Dotted straight lines show the margins for ***extreme values*** (three times the quartile distance from the lower or upper quartile [89:84]). At the bottom, the respective sample size n is given.

4.3.3 Characteristics of the individual measures

In this section, the measurements collected for EJABBERD will be characterized in regard to their location and spread. First the characteristics determined for the internal measures will be presented, then those for the external measures.

4.3.3.1 Characteristics of the internal measures

Table D.1, p. 117, and table D.2, p. 118, show the central moments of the internal measures for functions and modules. One can see that the distributions of all measures at function level and almost all measures at module level are clearly right-skewed and strongly concave [89:82f.]. This conforms to the results found by Ryder [80:342ff.] for HASKELL and also corresponds to observations of procedural and object-oriented programs [66:204] [90:92]. Among the measures which are to be examined when the hypotheses are tested, only the measures MAXCASE$_M$, MAXNEST$_M$, INSTF$_M$ and INSTM$_M$ show central moments which correspond roughly to those of a normal distribution [64:66]. Because of the marked asymmetry of most measures, with many "outliers", the median $\tilde{x}_{0.5}$ is in general more suitable as the measure of location than the arithmetic mean [17:82], and correspondingly the mean absolute deviation from the median $\tilde{d}_{0.5}$ as the measure of distribution [89:74].

Table D.3, p. 119, shows the minimum, maximum, quartile and outlier margins of the internal measures for functions. Lower outliers (in the box plot sense) do not occur (the margins are smaller than the theoretical and empirical minimum). While for some measures, very high extreme values occur, the majority of values lie in a small area around the median. The mean deviation from the median is generally much smaller than a tenth of the maximum. Table D.4, p. 120, shows the same characteristics for modules. Here there are also no lower outliers. The mean deviation from the median is generally larger than at function level, about a tenth of the respective maximum.

4.3.3.2 Characteristics of the external measures

Table D.5, p. 121, shows the central moments of the external measures at function level determined for EJABBERD. Almost all these measures are also clearly right-skewed and strongly concave. The measures for the mean processing time of issues, MEDITT(*)$_M$, form an exception. The same pattern can be seen for the external measures at module level, table D.6, p. 122.

Table D.7, p. 123, and table D.8, p. 124, show the minimum, maximum, quartile and outlier margins of the external measures for functions and modules. It is striking that except for the mean issue processing time, all the measures in the

Table 4.4 – Correlation matrix of the internal measures for functions in EJABBERD. Significant negative coefficients are shaded black, strong positive correlations are printed in **bold** and very strong ones additionally shaded grey. Correlation coefficients which are not highlighted are insignificant.

	$WMC(CYC)_M$	CBO_M	RFC_M	$INSTF_M$	$INSTM_M$	$MAXCASE_M$	$MAXNEST_M$	$NUMFUN_M$	$AVGLOC_M$	$AVGCALLS_M$
CBO_M	0.14	-								
RFC_M	**0.92**	0.15	-							
$INSTF_M$	**0.20**	-0.29	**0.25**	-						
$INSTM_M$	0.09	-0.07	0.14	**0.72**	-					
$MAXCASEM_M$	**0.58**	0.19	**0.62**	0.18	0.09	-				
$MAXNESTM_M$	**0.57**	0.17	**0.61**	**0.21**	0.10	**0.92**	-			
$NUMFUN_M$	**0.82**	0.15	**0.92**	0.17	0.06	**0.50**	**0.49**	-		
$AVGLOC_M$	**0.32**	0.02	**0.30**	0.16	0.16	**0.38**	**0.37**	0.11	-	
$AVGCALLS_M$	0.03	**0.47**	0.06	-0.48	-0.32	**0.24**	**0.23**	-0.02	**0.21**	-
LOC_M	**0.96**	0.12	**0.96**	**0.22**	0.10	**0.61**	**0.60**	**0.89**	**0.38**	0.04

area between the lower and upper quartile consistently have the value zero. This results from the low number of ·issues compared with the number of modules and functions which can be assigned to them (see section 4.2.3, p. 57).

4.3.4 Connections between the measures

The connections which were observed in this study between the measures which are used in the hypotheses will now be explained, first for the internal measures, then for the external ones.

4.3.4.1 Connections between the internal measures

One aspect of the criterion ·improvement validity is the question of whether a measure is more useful than the simplest measure, LOC_{FM}. In the literature, it is repeatedly stated that many software measures correlate strongly positively with LOC_{FM} [36:654] [44:6]. It is therefore not surprising that this is also the case in this study, as table 4.4 shows.

The measures $WMC(CYC)_M$, RFC_M and $NUMFUN_M$ correlate very strongly positively ($r > 0.80$) with the number of lines and with each other. One might expect that all three measures lead to very similar results when the hypotheses are tested. In case not all the expressions of a program are written in one line, new ·predicate nodes in control flow graphs (whose number is basically given by

Table 4.5 – Correlation matrix of the internal measures for functions of EJABBERD. Strong negative coefficients are shaded black, strong positive correlations are set in `bold`, and very strong ones are additionally shaded grey. Not highlighted correlation coefficients are negligible.

	FANOUT_F	MAXCASE_F	MAXNEST_F
MAXCASE_F	**0.58**	-	
MAXNEST_F	**0.66**	**0.89**	
LOC_F	**0.76**	**0.58**	**0.59**

$\text{WMC}(\text{CYC})_\text{M}$) can only arise through the addition of lines; the same is true for new functions or function calls (whose number is basically given by NUMFUN_M or RFC_M). Therefore, the correlation is based on a causal connection.

MAXCASE_M and MAXNEST_M also correlate strongly positively ($r \approx 0.60$) with the number of lines. This is due to the fact that with "normal" formatting, every nesting level uses additional lines.

The following are not significantly correlated with LOC_M ($r \leq 0.20$): the number of the modules coupled with a module, CBO_M, the ·instability measure INSTM_M and the average number of function calls per function of a module, AVGCALLS_M. This makes them interesting as measures which essentially represent something other than the size of a module. AVGCALLS_M and CBO_M are both moderately positivly correlated ($r = 0.47$), because they are both derived from the number of function call relationships. For the same reason, they are both moderately negatively correlated ($-0.48 \leq r \leq -0.29$) with the two ·instability measures, whose ·measurement functions have the number of call relationships in the denominator.

On the function level, all examined measures correlate strongly ($r > 0.5$) with the number of lines and with each other, as table 4.5 shows. FANOUT_F correlates most strongly with LOC_F, because with the increasing number of lines, the number of function calls must increase, unless mainly arithmetical or logical operations without function calls are carried out. For MAXCASE_F and MAXNEST_F, the above statement for the module level is valid.

Validation criterion ·factor independence ·Factor independence refers to the requirement that the components of derived measures should be independent of each other (see section 2.3.1.1, p. 29). This will now be investigated on the basis of linear correlation for the measures used in the hypotheses.

CBO_M can be understood as a measure derived from the number of efferent and afferent module couplings, OUTMODS_M and INMODS_M. The two measures show a significant correlation of $r = 0.31$[91]. This is not mainly due to a common connection with the number of lines, which is slightly negatively correlated with

[91] Confidence interval: $[0.27, 0.34]$

InMods$_M$ ($r = -0.04$ in the confidence interval $[-0.08, 0.00]$), whereas OutMods$_M$ is moderately positively correlated ($r = 0.40$ in the confidence interval $[0.38, 0.44]$).

The measure RFC$_M$ is defined as the sum of NumFun$_M$ und CallsOut$_M$. These two measures are strongly positively correlated, with $r = 0.86$[92], which is not surprising as with the increasing number of functions in the module, the number of function calls from this module must also increase, unless these functions merely carry out trivial calculations without external functions.

The ·instability measure InstM$_M$ is derived from the measures OutMods$_M$ and InMods$_M$, which have already been considered above. The variant InstF$_M$, by contrast, is derived from OutFuns$_M$ and InFuns$_M$. These show a similar correlation with LOC$_M$ as their module pendants: OutFuns$_M$ is strongly positively correlated ($r = 0.91$, confidence interval: $[0.90, 0.91]$), InFuns$_M$ is almost uncorrelated ($r = 0.023$, confidence interval: $[-0.01, 0.06]$).

4.3.4.2 Connections between the external measures

Table 4.6 on the following page shows the linear correlations between the external measures at module level for ejabberd. It is striking that in the modules affected by Errors[93], EndRate$_M$ correlates strongly ($r = 0.63$) with NumIss(B)$_M$. Bijlsma [8:39] defines EndRate(B)$_M$ as a measure of "project productivity" and interprets high values as a sign of efficient issue processing. However, on the basis of the observations of ejabberd, it can now be established that the measure EndRate(B)$_M$ is apparently mainly influenced by the Error content of the module. This is plausible, as a large number of Errors can only be solved when a large number of Errors actually exist. Thus, EndRate(B)$_M$ captures less the ·maintainability of the software than its Error-proneness. This will affect all the hypotheses in which EndRate(B)$_M$ is used as a measure of ·maintainability.

Furthermore, there are remarkable weak to moderate negative correlations (with $-0.43 \leq r \leq 0.20$) between the median processing time of issues, MedITT(B)$_M$, and the total number of issues or measures which strongly correlate with it (shaded black in table 4.6 on the following page). Therefore, the issue processing time was on average ***shortest*** for the modules with the most Errors. Possible explanations for this are on the one hand, that the high number of Errors reflects a large proportion of easily-corrected errors and on the other hand, that many Errors mainly occur in frequently used and especially important modules, where they are processed with higher priority (see section 2.3.4.2, p. 33 for distorting influences on the distribution of Errors among the program components). These suppositions should be tested in further research.

[92] Confidence interval $[0,85, 0,97]$

[93] See section 2.2.4, p. 26.

Table 4.6 – Correlation matrix of external measures for modules. Strong negative coefficients are shaded black, strong positive correlations are set in `bold` and very strong ones are additionally shaded grey. Non-highlighted correlation coefficients are negligible.

	NumIss(B)$_M$	BugRate$_M$	MedITT(B)$_M$	BMI$_M$	StartRate(B)$_M$	EndRate(B)$_M$
BugRate$_M$	**0.55**	-				
MedITT(B)$_M$	-0.20	0.00	-			
BMI$_M$	**0.34**	**0.26**	-0.28	-		
StartRate(B)$_M$	**0.76**	**0.43**	-0.43	0.23	-	
EndRate(B)$_M$	**0.74**	**0.40**	-0.37	**0.39**	**0.73**	-
ImproveRate$_M$	0.15	0.01	-0.03	**0.56**	0.10	0.07

4.3.5 Procedures used

The statistical procedures were selected according to the recommendations of the IEEE [46:18f.]. All the calculations are carried out with the statistics system R (see appendix B.1, p. 109). The results are rounded to two decimal places.

The ·validating criterion ·association is tested with Bravais-Pearson's correlation coefficient [89:131] (referred to below as r)[94], with the decision about the hypotheses being made on the basis of the 95 per cent confidence interval.

·Trackability is tested with Spearman's rank correlation coefficient [89:126] (referred to below as R). If the criterion ·association is already met, the criterion ·trackability is also met; the reverse is not the case.

Pearson's χ^2 statistic is used to test the criterion ·discriminative power. In the interests of comparability, the χ^2 results are presented normalized as absolute values of the Φ-coefficient in the interval $[0, 1]$ [89:112]. For independent features, $\Phi = 0$; the larger $\bar{\Phi}$ is, the stronger is the correlation. Only the threshold values given in the literature will be tested. On the basis of the data of several projects, better threshold values could be found, which does not seem feasable on the basis of a single project. The probability β of a ·type II error is calculated according to Cohen [18].

The number of errors is measured by NumIss(B)$_{FM}$. As margins for "significantly more" errors, the margins for outliers and extreme values according to box plot

[94] This is done according to Motulsky [70:300], in spite of the generally clearly non-normal distributions, under the assumption that the samples are large enough to give reliable results, based on the central limit theorem.

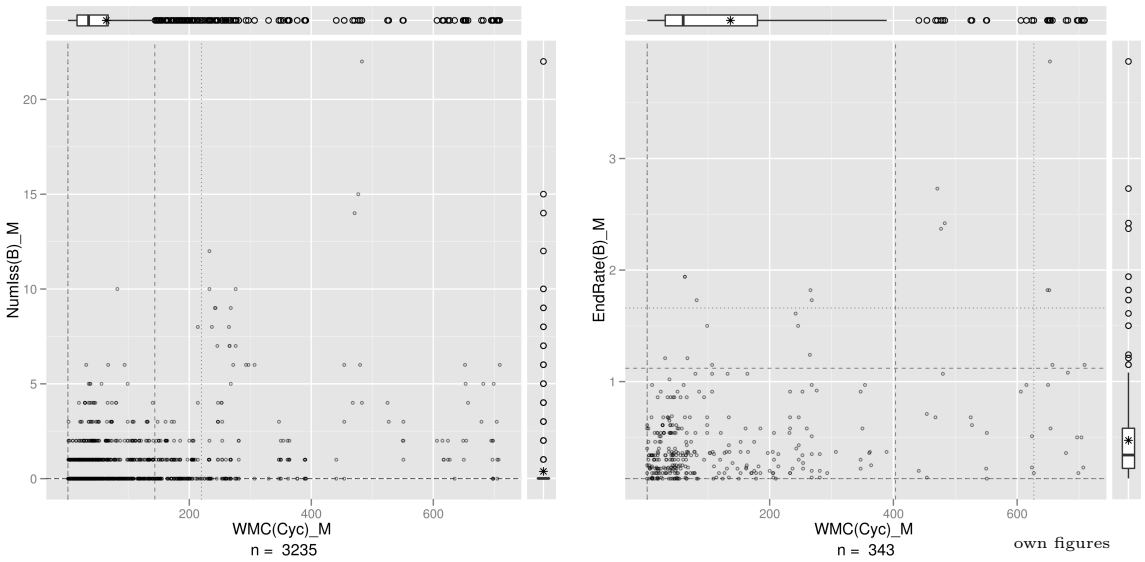


Figure 4.7 – Scatter plots for WMC(CYC)_M and $\text{NUMISS(B)}_\text{M}$ (left) and $\text{ENDRATE(B)}_\text{M}$ (right).

conventions were selected. As table D.7, p. 123 und table D.8, p. 124 show, these margins are all at zero, with a mean deviation from the median at 0.1 or 0.4 (rounded to two decimal places). On this basis, zero or more ERRORS were valued as "significantly more" for the contingency tables.

With these simple statistical procedures, only the ***linear*** correlation between measures is determined, and only the ·discriminative power in relation to ***given*** threshold values is tested.

4.3.6 Testing the hypotheses

4.3.6.1 Hypothesis AP1

Sub-hypothesis AP1.1 Figure 4.7 (left-hand diagram) plots, for all the versions of all the modules, the corresponding WMC(CYC)_M value against the number of ERROR issues $\text{NUMISS(B)}_\text{M}$ which affect the respective version of the module.

The two box plots make the significant right-skew of the marginal distributions evident, which will also be observed with the other measures: most of the modules have no or only a few ERRORS (albeit over their entire life-span!) and a WMC(CYC)_M value under 200. The box of the box plot for $\text{NUMISS(B)}_\text{M}$ is hardly recognizable as more than half of the modules have no ERRORS at all. Several significantly more "weighty" modules have less than five ERRORS. By contrast, larger $\text{NUMISS(B)}_\text{M}$ values mainly occur in modules over 200 WMC(CYC)_M.

In total, the linear correlation is $r = 0.38$,[95] which means that the null hypothesis is rejected in favour of hypothesis AP1.1.

Sub-hypothesis AP1.2 Figure 4.8, p. 76 (left-hand diagram) shows the values of $\text{WMC}(\text{Cyc})_\text{M}$ of all versions of all modules with the corresponding value of BugRate_M, the percentage of Errors in all the issues which affect the respective module version.

As more than half of the modules have no Errors, the box for the distribution of BugRate_M is compressed to the zero point. This means that those module versions predominate which contain no or only very few Errors or other issues. In view of the fact that most modules do not have more than four Errors (see fig. 4.7 on the previous page), the accumulations at 0.5 and 1 for BugRate_M point to many modules with one or two Errors among just as many other issues.

The highest proportions of Errors occur in the modules with $\text{WMC}(\text{Cyc})_\text{M}$ values below 200, while in the middle range of the spectrum, hardly any proportions over 0.5 occur and none at all occur among the "weightiest" modules. This concurs with the observations of other authors, that larger modules contain relatively few errors, which is explained by the greater attention and care paid to the processing of larger modules [cf. 44:9f.].

Overall, over the whole range of values there is only a linear correlation of $r = 0.17$, which is not sufficient to reject the null hypothesis for the criterion ·association. However, the rank correlation coefficient is significant with $p = 0.00$ and weak with $R = 0.14$, but for the criterion ·trackability it is sufficient.

Sub-hypothesis AP1.3 $\text{WMC}(\text{Cyc})_\text{M}$ shows a significant rank correlation ($p = 0.14$) with the median processing time of Errors in hours, $\text{MedITT}(\text{B})_\text{M}$. However, the strength of the correlation, $R = 0.14$, has to be rated as negligible. As Pearson's correlation coefficient is not significant ($p = 0.93$), the null hypothesis is upheld with regard to both criteria, ·association and ·trackability.

Sub-hypothesis AP1.4 The correlation of $\text{WMC}(\text{Cyc})_\text{M}$ with the ·Backlog Management Index BMI_M is also negligible. The distribution of the value pairs shows no recognizable pattern (see fig. D.1, p. 119 in the Appendix). However, counter to the postulated negative correlation, there was a positive correlation of $r = 0.17$[96] here. The null hypothesis is upheld against hypothesis AP1.4.

[95] Confidence interval: $[0.35, 0.41]$
[96] Confidence interval: $[0.14, 0.21]$

Sub-hypothesis AP1.5 The measure $\text{WMC}(\text{CYC})_\text{M}$ shows a positive correlation with $\text{ENDRATE}(\text{B})_\text{M}$, where a negative correlation was surmised. Figure 4.7, p. 73 (right-hand diagram) shows the $\text{WMC}(\text{CYC})_\text{M}$ values of all module versions which contain ERRORS, with the corresponding average number of solved ERRORS per month during the life-span of the version. The distribution is similar to that in the left-hand diagram. $\text{ENDRATE}(\text{B})_\text{M}$ is also strongly correlated with $\text{NUMISS}(\text{B})_\text{M}$.

With $r = 0.34$[97], the linear correlation with $\text{WMC}(\text{CYC})_\text{M}$ is somewhat weaker than that of $\text{NUMISS}(\text{B})_\text{M}$. The result indicates that the basic assumption which led to the formulation of hypothesis AP1.5 – that $\text{ENDRATE}(\text{B})_\text{M}$ correlates positively with maintainability – is too simple. The influence of the number of ERRORS seems to prevail. Hypothesis AP1.5 is withdrawn because it does not seem to be rationally justified. But the result is also not compatible with the null hypothesis. Further research on the relationship between $\text{ENDRATE}(\text{B})_\text{M}$ and ·maintainability seems appropriate.

Sub-hypothesis AP1.6 For modules, the measure for the percentage of IMPROVEMENTS in all the issues, $\text{IMPROVERATE}_\text{M}$ (fig. 4.8 on the following page, right-hand diagram), shows the same accumulations at 0, 0.5 and 1 in the lower value range of $\text{WMC}(\text{CYC})_\text{M}$ as BUGRATE_M, with the marginal distribution significantly more evenly spread, as the box spanning more than two-thirds of the domain shows. In the whole domain of $\text{WMC}(\text{CYC})_\text{M}$, minimum, medium and maximum $\text{IMPROVERATE}_\text{M}$ values occur evenly.

Counter to the hypothetic negative correlation, there is a positive though negligible correlation coefficient of $r = 0.10$.[98] With $p = 0.00$, the rank correlation is significant, but contrary to expectations it is considerably positive: $R = 0.22$. This leads to upholding the null hypothesis that there is no significant negative correlation, against hypothesis AP1.6.

The fact that with regard to $\text{WMC}(\text{CYC})_\text{M}$ larger modules do not, as surmised, show lower improvement rates, can be partly explained by the lower error density – modules with less errors to be resolved offer relatively more space for improvements. Overall, hypothesis AP1.6 has to be rejected.

Summary Overall, the sub-hypotheses of AP1 have proved to be incompatible with each other, which is mainly due to the fact that connections between the external measures appeared, which were not anticipated when the hypotheses were put forward. For this reason, a summary statement on hypothesis AP1 cannot be made.

[97] Confidence interval: $[0.24, 0.43]$
[98] Confidence interval: $[0.04, 0.16]$

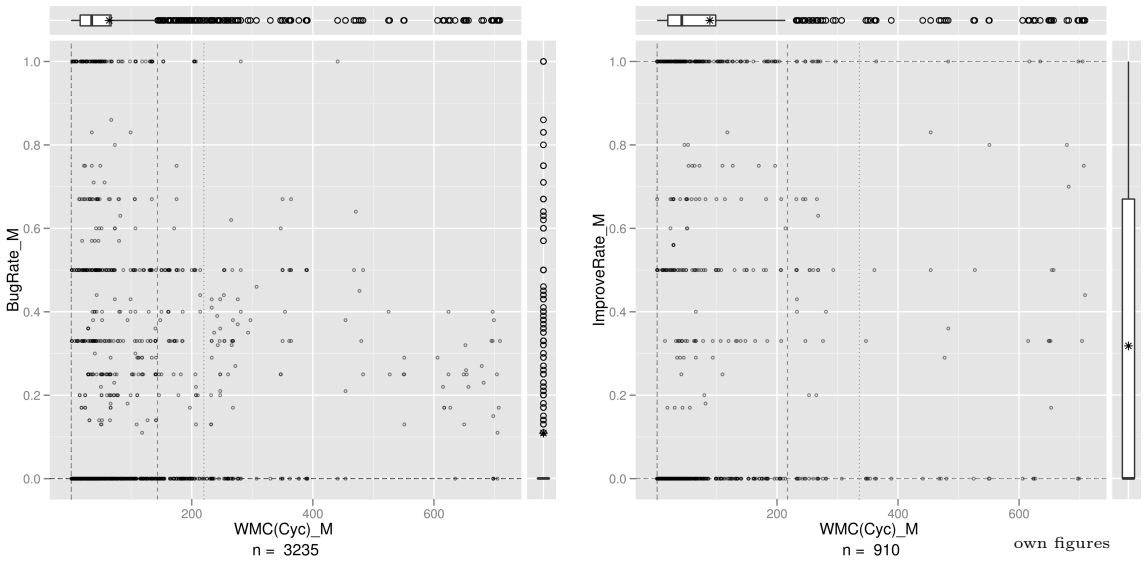


Figure 4.8 – Scatter plots for WMC(CYC)_M and BUGRATE_M (left) and $\text{IMPROVERATE}_\text{M}$ (right).

It has been established that WMC(CYC)_M correlates moderately positively with the number of ERRORS ($r = 0.38$) and, counter to expectations, with the rate of resolving ERRORS ($r = 0.34$). Further, there is a weak rank correlation with the percentage of ERRORS in the issues ($R = 0.29$) and with the percentage of IMPROVEMENTS in the resolved issues ($R = 0.22$).

4.3.6.2 Hypothesis AP2

Sub-hypotheses AP2.1 and AP2.2 With regard to the measures BUGRATE_M and $\text{NUMISS(B)}_\text{M}$, CBO_M shows practically no linear correlation ($r = 0.05$ or $r = 0.06$); therefore, the null hypothesis is upheld in both cases.

While the distribution for BUGRATE_M gives no evidence of any marked correlation (see fig. D.2, p. 125), the distribution for $\text{NUMISS(B)}_\text{M}$ (fig. 4.9 on the following page, right-hand diagram) shows a marked accumulation of especially ERROR-prone modules at a CBO_M value around 25, while for the especially strongly-coupled modules (for instance, from a CBO_M value around 50), a slight accumulation of relatively ERROR-prone modules with a value around 70 is evident. Further research could consider the peculiarities of these modules in more detail.

Sub-hypothesis AP2.3 Figure 4.9 on the next page shows the correlation between the coupling measure CBO_M and the median processing time of ERRORS, $\text{MEDITT(B)}_\text{M}$, where every point corresponds to a module affected by ERRORS in a particular version.

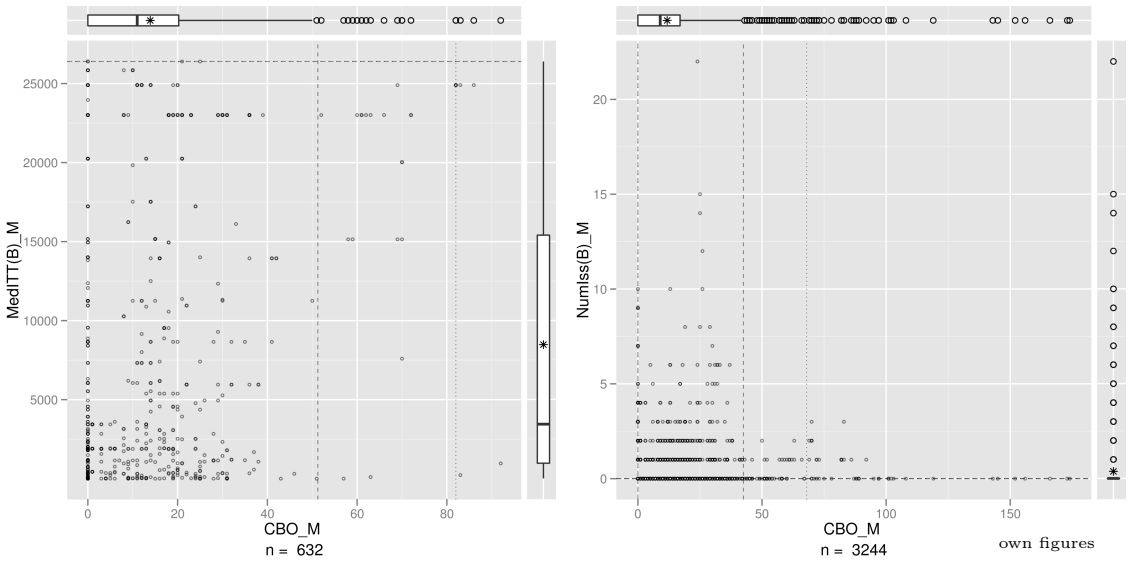


Figure 4.9 – Scatter plots for $\mathrm{CBO_M}$ and $\mathrm{MEDITT(B)_M}$ (left), or $\mathrm{NUMISS(B)_M}$, respectively (right)

While in the lower range of $\mathrm{CBO_M}$ (under 40) all processing times occur, from almost zero to over 25 000 hours (which means almost three years), in the upper range (over 40) there is a clear positive tendency.

Together, this results in a weak Pearson's correlation coefficient of $r = 0.22$, which due to the confidence interval $[0.14, 0.29]$ is not sufficient to reject the null hypothesis, not least because the rank correlation is different from 0, but also too weak: $R = 0.12$. The null hypothesis must be upheld.

Sub-hypotheses AP2.4 to AP2.6 $\mathrm{CBO_M}$ shows no recognizable correlation with the measures $\mathrm{BMI_M}$, $\mathrm{ENDRATE_M}$ and $\mathrm{IMPROVERATE_M}$, either. The correlation coefficients are negligible ($-0.05 \leq r \leq 0.16$, $0,00 \leq R \leq 0.07$). The null hypotheses are upheld.

Summary For the measure $\mathrm{CBO_M}$, none of the hypotheses could be confirmed. A significantly positive or negative correlation was not established with any of the indicators adopted for ·maintainability.

4.3.6.3 Hypothesis AP3

Sub-hypothesis AP3.1 Figure 4.10 on the following page presents the correlation of the measure for ·response for a class, $\mathrm{RFC_M}$, in all versions of all modules with the corresponding ERROR counts ($\mathrm{NUMISS(B)_M}$, left-hand diagram) or the

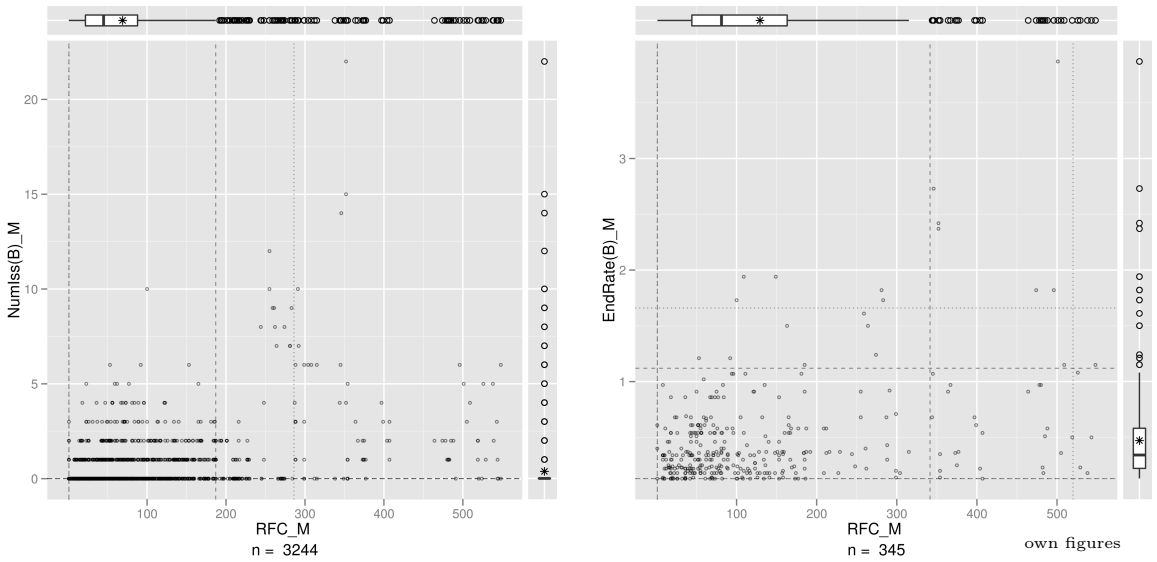


Figure 4.10 – Scatter plots for $\mathrm{RFC_M}$ and $\mathrm{NumIss(B)_M}$ (left), and $\mathrm{EndRate(B)_M}$ (right)

average number of Errors resolved per month during the life-span of the versions ($\mathrm{EndRate(B)_M}$, right-hand diagram).

As was already established above, these two external measures correlate strongly with each other. Their common distributions with $\mathrm{RFC_M}$ are also similar. A very strong accumulation of modules with less than five Errors can be identified at $\mathrm{RFC_M}$ values below 200. Modules between 200 and 400 $\mathrm{RFC_M}$ show a much larger Error content, while modules with an $\mathrm{RFC_M}$ value around 500 are rather similar to those in the lower value range, without however the extreme accumulation of modules without Errors.

Overall, there is a moderate linear correlation (the strongest in this study) at $r = 0.39$,[99] so that the null hypothesis is rejected in favour of hypothesis AP3.1.

Sub-hypotheses AP3.2 and AP3.5 $\mathrm{RFC_M}$ does not show a sufficient Pearson correlation with the measures for the Error ratio of all open issues, $\mathrm{BugRate_M}$, and for the share of Improvements among all resolved issues, $\mathrm{ImproveRate_M}$.[100] For this reason, the null hypothesis in relation to the criterion ·association is upheld. However, both measures show a sufficient, weak rank correlation ($R = 0.29$ and $R = 0.25$), so that the null hypotheses under the criterion ·trackability are rejected in favour of the hypotheses AP3.2 and AP3.5. The common distributions of $\mathrm{BugRate_M}$ and $\mathrm{ImproveRate_M}$ with $\mathrm{RFC_M}$ resemble those with $\mathrm{WMC(Cyc)_M}$.

[99] Confidence interval: $[0.36, 0.42]$

[100] $r = 0.18$ for $\mathrm{BugRate_M}$, $r = 0.15$ for $\mathrm{ImproveRate_M}$.

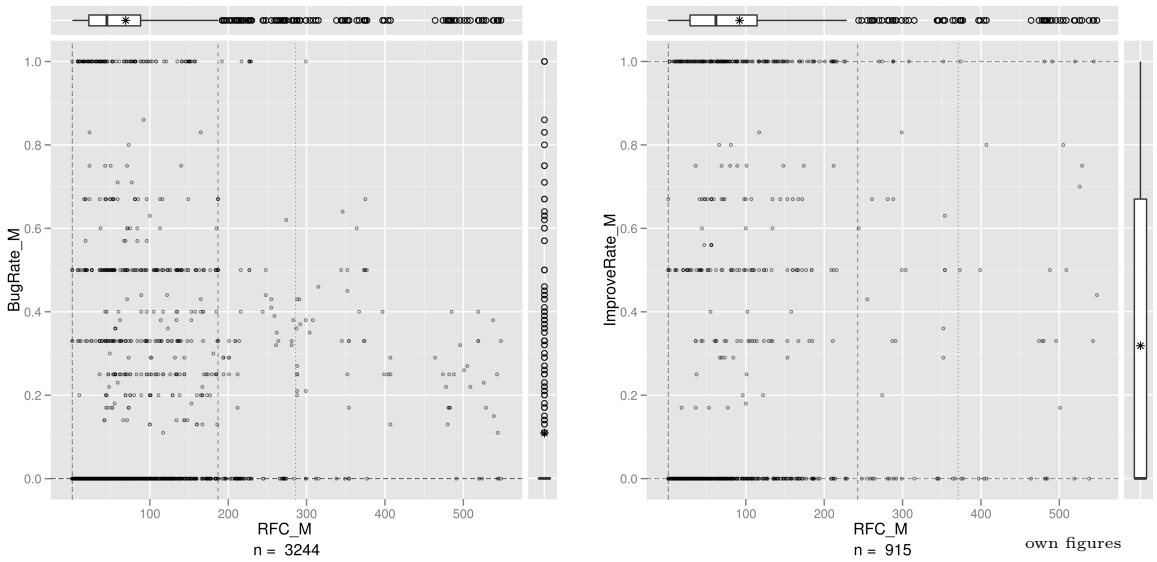


Figure 4.11 – Scatter plots for $\mathrm{RFC_M}$ and $\mathrm{BUGRATE_M}$, or $\mathrm{IMPROVERATE_M}$, respectively

Sub-hypothesis AP3.3 Compared with $\mathrm{WMC(CYC)_M}$, $\mathrm{RFC_M}$ shows a somewhat stronger positive correlation with the measure $\mathrm{BMI_M}$: $r = 0.20$.[101] The hypothesis of a negative correlation with absolute value > 0.20 must be rejected. In view of the confidence interval, the null hypothesis that the value of the correlation coefficient is ≤ 0.20, is not refuted and will therefore be upheld. The common distribution of $\mathrm{RFC_M}$ with $\mathrm{BMI_M}$ is the same as that for $\mathrm{WMC(CYC)_M}$ and shows no notable regularity (see fig. 4.12 on the next page).

Sub-hypothesis AP3.4 Contrary to expectations, $\mathrm{CBO_M}$ also shows a moderate positive correlation with $\mathrm{ENDRATE(B)_M}$: $r = 0.36$.[102] It has already been explained above that $\mathrm{ENDRATE(B)_M}$ is probably only marginally a measure for ·maintainability and that the hypothesis is therefore no longer rationally justified.

Summary For hypothesis AP3, too, no overall evaluation can be given, as under the basic assumptions (see section 4.3.1.1, p. 63) the individual results are contradictory or contradict them.

It can be established that $\mathrm{RFC_M}$ correlates moderately positively ($r = 0.39$) with the number of ERRORS and, against expectations, with the rate of solution of ERRORS per month ($r = 0.36$). Weak rank correlations exist with the ratio of ERRORS among all issues ($R = 0.29$) and the ratio of IMPROVEMENTS among resolved issues ($R = 0.25$).

[101] Confidence interval: $[0.16, 0.23]$
[102] Confidence interval: $[0.27, 0.45]$

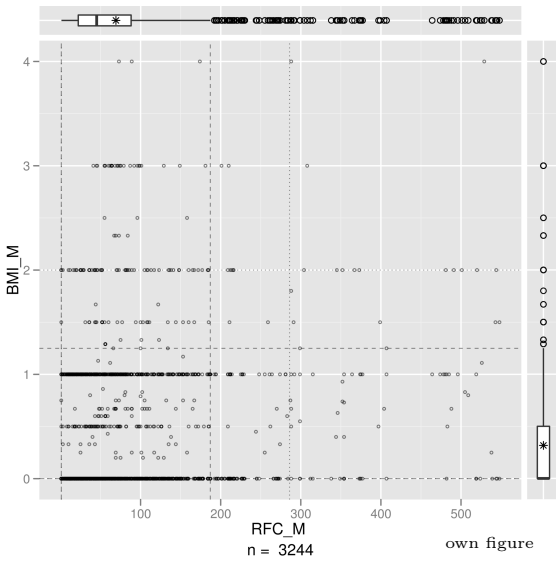


Figure 4.12 – Scatter plot of $\mathrm{RFC_M}$ and $\mathrm{BMI_M}$

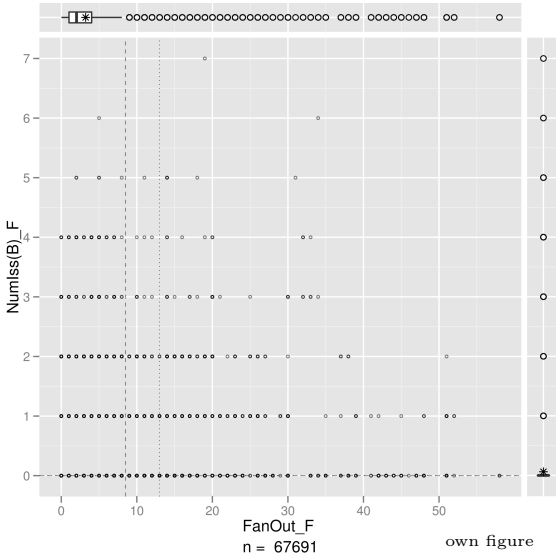


Figure 4.13 – Scatter plot of $\mathrm{FanOut_F}$ and $\mathrm{NumIss(B)_F}$

4.3.6.4 Hypothesis AP4

Contrary to the report by Ryder [80:148f.] of a moderate correlation, with $r \approx 0.5$ ($p \leq 0.05$), of the ·fan-out of functions with the number of error corrections, $\mathrm{FanOut_F}$ showed only a negligible linear correlation with the number of Errors: $r = 0.16$, with a confidence interval $[0.15, 0.17]$. Figure 4.13 seems to show a negative correlation for the outliers (according to box plot convention). In case this does exist, it will be overall overlain by the overwhelming concentration of functions without Errors (evident in the box for the marginal distribution box

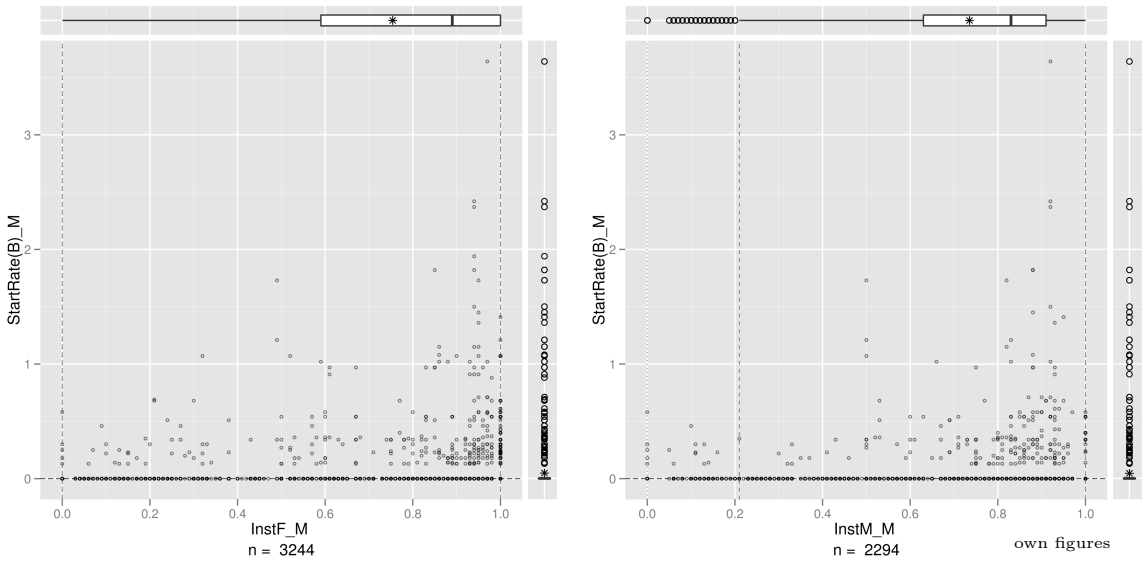


Figure 4.14 – Scatter plots for INSTF_M or INSTM_M, respectively, with $\text{STARTRATE(B)}_\text{M}$

plot, compressed to zero for $\text{NUMISS(B)}_\text{F}$. Hypothesis AP4 has to be rejected.

4.3.6.5 Hypothesis AP5

The measure for the ·instability of modules, with the two variants INSTF_M and INSTM_M, shows practically no correlation overall with the average number of new ERRORS per month through the life-span of the modules, $\text{STARTRATE(B)}_\text{M}$: in both cases $r = 0.07$. However, in fig. 4.14 it is evident that the correlation for modules whose $\text{STARTRATE(B)}_\text{M}$ value is not zero seems to be much stronger. The overwhelming influence of the large number of ERROR-free modules, for which $\text{STARTRATE(B)}_\text{M}$ is zero, is again evident in the boxes for the marginal distributions compressed to zero. Further studies are needed to establish whether a more complete assignment of ERRORS to program components would significantly reduce this influence.

Hypothesis AP6 Hopkins and Hatton [44:8] (see section 5.3, p. 88) report a weak but significant ***negative*** correlation between the maximum nesting depth of `if` branching and the number of errors in the corresponding functions of a large FORTRAN library – which contradicts common programming guidelines. Figure 5.1, p. 89 shows the diagram that presents the negative correlation at function level which they ascertained. Contrary to these results (and in compliance with the programming guidelines mentioned) the present research did not lead to a negative but a positive correlation at function level, which, however, does not reach

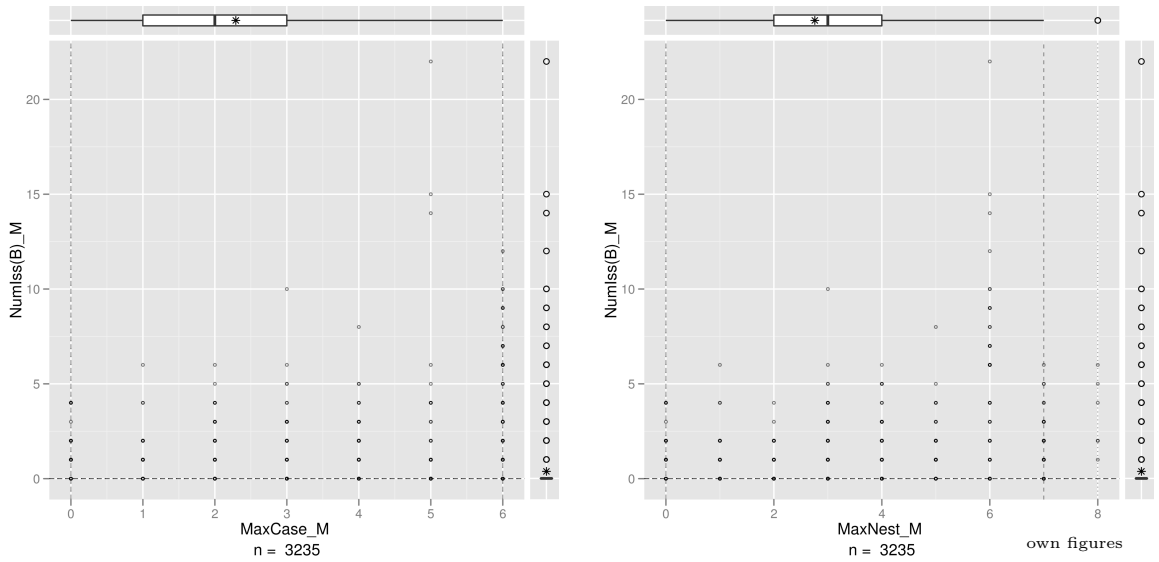


Figure 4.15 – Scatter plots for $\text{MaxCase}_{\text{M}}$ and $\text{NumIss(B)}_{\text{M}}$ as well as $\text{MaxNest}_{\text{M}}$ and $\text{NumIss(B)}_{\text{M}}$

the required effect size.[103] Nevertheless, the null hypothesis will be upheld, as hypothesis AP6 was not confirmed.

At module level, however, a moderate ***positive*** correlation both between the maximum nesting depth of `case` expressions alone ($\text{MaxCase}_{\text{M}}$) and also for `begin`/`end`, `case`, `fun`, `if` and `receive` expressions ($\text{MaxNest}_{\text{M}}$) was observed: $r = 0.29$ and $r = 0.27$.[104] This positive correlation at module level is clearly recognizable in fig. 4.15.

4.3.6.6 Hypothesis T1

For the two sub-hypotheses, the maximum nesting depth of `case` expressions alone ($\text{MaxCase}_{\text{FM}}$) and of `begin`/`end`, `case`, `fun`, `if` and `receive` expressions ($\text{MaxNest}_{\text{FM}}$) was considered, for the function and the module level respectively (identified by the indices F and M). Table 4.7 on the next page shows the corresponding contingency tables.

The Φ coefficient takes (in the reading order of the tables) the low values 0.1, 0.25, 0.1 and 0.23, with p and $\beta = 0.00$. The correlation seems to be more strongly marked at module level than at function level. A possible reason for this is that the assignation of issues to program components at module level is more complete and thus the excess weight of Error-free components drops. The null hypothesis of the independence of the features is rejected both for $\text{MaxCase}_{\text{FM}}$ and also for

[103] Both for $\text{MaxCase}_{\text{F}}$ as well as for $\text{MaxNest}_{\text{F}}$: $r = 0.13$. See also fig. D.3, p. 126.
[104] Confidence interval: $[0.26, 0.32]$ and $[0.24, 0.30]$, respectively.

Table 4.7 – Contingency tables for $\text{MaxCase}_{\text{FM}}$ and $\text{MaxNest}_{\text{FM}}$, respectively, and $\text{NumIss(B)}_{\text{FM}}$

		$\text{MaxCase}_{\text{F}}$	
		≤ 2	> 2
$\text{NumIss(B)}_{\text{F}}$	$= 0$	64979	3259
	> 0	2885	510

		$\text{MaxCase}_{\text{M}}$	
		≤ 2	> 2
$\text{NumIss(B)}_{\text{M}}$	$= 0$	1758	961
	> 0	225	450

		$\text{MaxNest}_{\text{F}}$	
		≤ 2	> 2
$\text{NumIss(B)}_{\text{F}}$	$= 0$	61848	6390
	> 0	2592	803

		$\text{MaxNest}_{\text{M}}$	
		≤ 2	> 2
$\text{NumIss(B)}_{\text{M}}$	$= 0$	1422	1297
	> 0	157	518

Table 4.8 – Contingency table for LOC_{M} and $\text{NumIss(B)}_{\text{M}}$

		LOC_{M}	
		≤ 400	> 400
$\text{NumIss(B)}_{\text{M}}$	$= 0$	2164	561
	> 0	325	353

$\text{MaxNest}_{\text{FM}}$ (that is, at both function and module level) in favour of hypothesis T1.

4.3.6.7 Hypothesis T2

With the contingency table 4.8, $\bar{\Phi} = 0.28$ with p and $\beta = 0.00$. Thus the null hypothesis is rejected in favour of hypothesis T2, although the correlation is weak.

4.3.6.8 Hypothesis T3

As the threshold value for this hypothesis is given as an interval, the lower and upper margins will be tested separately. For the contingency table 4.9, $\bar{\Phi} = 0.21$ and $\bar{\Phi} = 0.29$, respectively, with p and $\beta = 0.00$ respectively. The null hypothesis is rejected both for the upper and also for the lower margins in favour of hypothesis T3.

Table 4.9 – Contingency tables for NumFun_{M} and $\text{NumIss(B)}_{\text{M}}$

		NumFun_{M}	
		≤ 30	> 30
$\text{NumIss(B)}_{\text{M}}$	$= 0$	2215	510
	> 0	400	278

		NumFun_{M}	
		≤ 40	> 40
$\text{NumIss(B)}_{\text{M}}$	$= 0$	2527	198
	> 0	471	207

Table 4.10 – Contingency tables for LOC_F and $\text{NumIss(B)}_\text{F}$

		LOC_F	
		≤ 15	> 15
$\text{NumIss(B)}_\text{F}$	$= 0$	49137	19101
	> 0	1615	1780

		LOC_F	
		≤ 20	> 20
$\text{NumIss(B)}_\text{F}$	$= 0$	54578	13660
	> 0	1916	1479

Table 4.11 – Contingency tables for AvgLOC_M and $\text{NumIss(B)}_\text{M}$

		AvgLOC_M	
		≤ 28.65	> 28.65
$\text{NumIss(B)}_\text{M}$	$= 0$	2543	176
	> 0	617	58

		AvgLOC_M	
		≤ 37.5	> 37.5
$\text{NumIss(B)}_\text{M}$	$= 0$	2622	97
	> 0	649	26

4.3.6.9 Hypothesis T4

Here again the lower and upper margins of the threshold interval will be considered separately (see table 4.10). The correlation is significant (p and $\beta = 0.00$ respectively), even if with $\bar{\Phi} = 0.11$ and $\bar{\Phi} = 0.12$ relatively weak. The null hypothesis must be rejected.

4.3.6.10 Hypothesis T5

Two variants of a threshold value for the measure AvgLOC_M are tested to find out whether the error content of the corresponding modules is significantly different (see table 4.11). The hypothetical threshold value on the basis of the median and the mean deviation from the median (hypothesis T5.1) is $1.5 \cdot (\tilde{x}_{0.5} + \tilde{d}_{0.5}) = 28.65$, the one on the basis of the arithmetic mean and the standard deviation (hypothesis T5.2) is $1.5 \cdot (\bar{x} + s) = 37.5$. In the first case, the result is just a little below significance ($\bar{\Phi} = 0.03$, $p = 0.06$, $\beta = 0.00$), in the second case it is clearly not significant ($p = 0.81$, $\bar{\Phi} = 0.00$, $\beta = 0.00$), so that in both cases the null hypothesis is upheld against hypothesis T5.2.

Hypothesis T6 Here again the threshold value based on the median (6.3) and also the threshold according to Marinescu and Lanza [60:29] (8.775) will be considered. Table 4.12 on the following page shows the corresponding contingency tables. There is a significant but weak correlation with the median-based threshold value ($\bar{\Phi} = 0.04$, $p = 0.03$, $\beta = 0.00$), so that the null hypothesis must be rejected in favour of hypothesis T6.1. The result for the other threshold value is not significant ($\bar{\Phi} = 0.03$, $p = 0.14$, $\beta = 0.00$), so that the null hypothesis is upheld against hypothesis T6.2.

Table 4.12 – Contingency tables for $\text{AvgCalls}_{\text{M}}$ and $\text{NumIss(B)}_{\text{M}}$

		$\text{AvgCalls}_{\text{M}}$	
		≤ 6.3	> 6.3
$\text{NumIss(B)}_{\text{M}}$	$= 0$	2522	203
	> 0	610	68

		$\text{AvgCalls}_{\text{M}}$	
		≤ 8.77	> 8.77
$\text{NumIss(B)}_{\text{M}}$	$= 0$	2614	111
	> 0	641	37

5 Results and discussion

In this chapter, the results of the foregoing research will be summarized and discussed.

5.1 Suitability of the external measures for evaluating maintainability

Some of the basic assumptions which have been made earlier, according to which certain external measures were assumed to be indicators for better or worse ·maintainability, are thrown into doubt by the present study. The average number of ERRORS resolved per month ($\text{ENDRATE(B)}_\text{M}$) was viewed as a sign of the fact that errors can be easily resolved. However, it became clear that this solution rate is closely connected with the total number of ERRORS. This result indicates that $\text{ENDRATE(B)}_\text{M}$ in the form used here cannot serve as a measure of ·maintainability.

The percentage of IMPROVEMENTS in all resolved issues ($\text{IMPROVERATE}_\text{M}$), which only correlated weakly positively with the number of ERRORS, was at least not refuted as an indicator of good ·maintainability.

The mean processing time of ERRORS ($\text{MEDITT(B)}_\text{M}$), which was expected to have larger values for less easily maintainable components, surprisingly shows a significant negative correlation with the number of ERRORS. As mentioned above, this is probably due not mainly to internal quality features but to the higher prioritization of components containing ERRORS during maintenance.

5.2 Connections between internal and external quality features

The results dealing with linear dependencies between internal and external measures are inconsistent. None of the main hypotheses which refer to the validating criteria ·association and ·trackability could be entirely adopted or rejected. The weaker hypotheses on the criterion ·discriminative power could be overwhelmingly confirmed.

5.2.1 Association and trackability

The total cyclomatic complexity of the functions of a module and the size of the ·response set ($\mathrm{WMC}(\textsc{Cyc})_\mathrm{M}$ or RFC_M), that is, the number of further function calls potentially triggered by a function call, showed the strongest correlation with the number of Errors,[105] the average number of Errors resolved per month,[106] and the ratio of Errors among all the issues of a module[107] and with the ratio of Improvements among the issues resolved in the issue tracking system.[108] The strong linear correlation between the two measures could point to a common background influence, which would explain the similarity of the results. This could not be checked in this study.

The strength of the ·coupling of a module in relation to other modules showed no significant correlation with any of the investigated measures for ·maintainability. This is contrary to the usual recommendations, according to which strong coupling of modules supposedly exerts a negative influence on their maintainability [cf. 5:474]. Similarly, it could not be confirmed that the ·instability of modules [61:262] corresponds to the frequency of modifications, measured by the occurrence of new ·issues in the issue tracking system. Ryder's observation of a strong correlation between the out-degree of functions, that is, the extent to which they call other functions, and the error content of functions [80:148] could not be confirmed.

The result found by Hopkins and Hatton [44:8], that deeply-nested case distinctions are associated with ***fewer*** errors, in contradiction to the usual recommendations, could not be reproduced. In any case, the authors explained this with external influences such as the greater care taken by developers, which is obviously not measured by internal software measures.

5.2.2 Discriminative power

There exists a weak correlation according to which, for functions and modules in which `begin`/`end`, `fun`, `if` or `receive` expressions are nested in three or more levels, many more error reports occur, while the nesting depth of `case` expressions shows no significant correlations.

The difference with regard to the number of lines and the number of functions of modules is somewhat stronger: modules which contain more than 400 lines or more than 30 or 40 functions, have many more error reports than those which are shorter or contain fewer functions.

[105] $r = 0.38$ or $r = 0.39$, respectively.
[106] $r = 0.34$ or $r = 0.36$, respectively.
[107] Each: $R = 0.29$.
[108] $R = 0.22$ or $R = 0.25$, respectively.

Table 5.1 – Linear correlation of the number of lines (LOC_M) with the examined external measures. The confidence intervals for BMI_M with $[0.16, 0.22]$ and for $BUGRATE_M$ with $[0.14, 0.21]$ are not sufficient for the classification of their respective correlations as considerable.

$NUMISS(B)_M$	$BUGRATE_M$	$MEDITT(B)_M$	BMI_M	$ENDRATE(B)_M$	$IMPROVERATE_M$
0.42	0.17	0.00	0.19	0.36	0.11

No significant differences in the number of ERRORS could be identified with a median-based threshold value of 15 to 20 lines for individual functions or with a threshold value based on the arithmetic mean of 28 to 38 lines on average per function of a module. However, the result for the median-based threshold was only just non-significant, while the second threshold value proved entirely useless.

With regard to the average number of different function calls per function of a module, a significant difference between modules on this side and that side of a median-based threshold was observed: modules with an average of more than 6.3 calls had significantly more ERRORS than modules with less calls.

5.2.3 Comparison of some results with LOC_{FM}

A comparison of the Pearson correlation coefficient for the connection between the number of lines (LOC_M) and the external measures tested in hypotheses AP1 to AP3 shows that, when only the linear correlation is considered, the measures $WMC(CYC)_M$ and RFC_M do not have any ·improvement validity compared with the number of lines – whose relation to the measures $NUMISS(B)_M$ and $ENDRATE(B)_M$, with reference to which hypotheses AP1.1 and AP1.5 or AP3.1 and AP3.4 were confirmed, is just as strong as or stronger than the corresponding relations of $WMC(CYC)_M$ and RFC_M. It would be useful to investigate whether significant differences in the three measures were evident if sub-sets of the sample or non-linear connections were taken into consideration. Until then, the added benefit of $WMC(CYC)_M$ and RFC_M is doubtful.

5.3 Comparison with the study of Hopkins and Hatton

Similarly to this study, in a case study with a large, mature[109] FORTRAN library, Hopkins and Hatton [44] found significant though weak correlations with the number of errors for 15 well-known software measures (see, for example, fig. 5.1 on

[109] 260000 lines of code, 3600 functions, 20 years of usage.

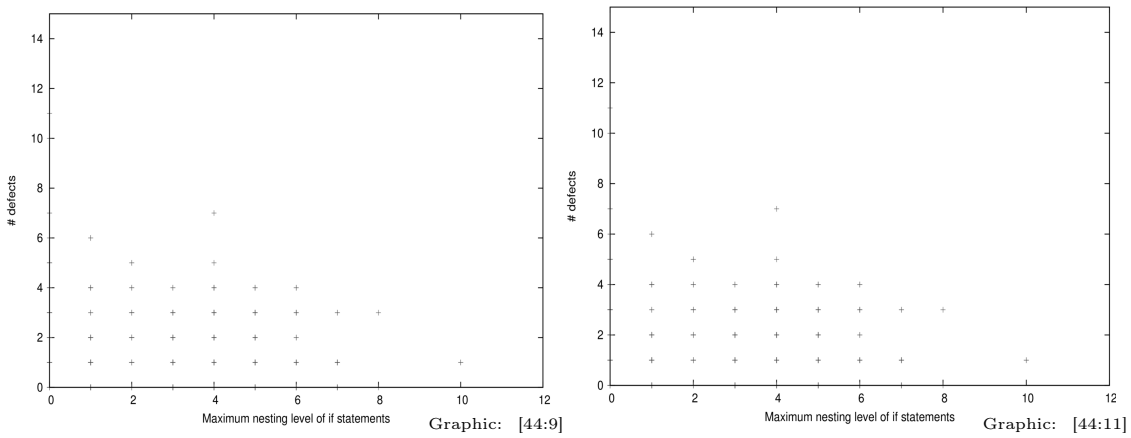


Figure 5.1 – Scatter plots from the study by Hopkins and Hatton [44]. In the left-hand diagram, the number of executable lines is plotted with the corresponding number of errors; in the right-hand diagram, with the maximum nesting depth of `if` instructions.

the next page). As discussed in section 2.3.4.2, p. 33, this may partly be due to the fact that individual components show different usage profiles and the number of known and reported errors is distorted by these usage differences. In order to compensate for this, the authors group the measurements of the internal measures according to their number of errors and investigate the connection between the number of errors and the arithmetic mean values of the groups [44:12]. In this way, they obtain much stronger correlations in the expected direction, but also more non-significant results because of the data reduction.

For lack of time, this approach could not be completely carried out for this study. As an example, fig. 5.2 on the next page should be noted, however, where the modules grouped according to the value of $\textsc{StartRate}_{\mathrm{M}}$ and the associated median values of the ·instability measure $\textsc{InstM}_{\mathrm{M}}$ are presented. The linear correlation according to Pearson is $r = 0.32$, although with a very large confidence interval of $[0.03, 0.55]$. In view of the reduced sample sizes, it is not surprising that there were many more non-significant results among some of the correlation coefficients determined in this way than without this transformation of the data. Further research is needed to clarify the usefulness of this procedure.

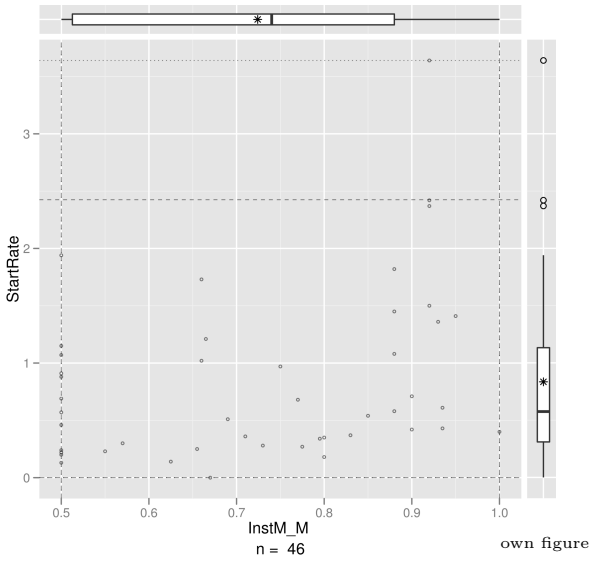


Figure 5.2 – Scatter plot of $\text{StartRate}_\text{M}$ and the median of the InstM_M values of all modules with a certain $\text{StartRate}_\text{M}$ value

6 Summary

This study is the first empirical study in the published German- and English-language literature on the validation of software measures for a functional programming language, ERLANG, which has a large professionally used software product as its research object. For a series of internal measures, the research served to prove significant connections with external quality features of the software investigated, EJABBERD. For the first time, as far as the author is aware, a series of programming guidelines for ERLANG could be empirically verified.

For the investigation of the internal and external measurements, a measuring environment was developed which can be used and further developed for future empirical research into ERLANG systems. The raw data, only a small part of which could be analysed for this study, can be made available for further research and would enrich central databases such as FLOSSMetrics[110] or PROMISE[111], which to date contain hardly any measurements of functional programs.

Finally, since for lack of alternatives for the static analysis of the research object, it was necessary to rely on a measuring tool (REFACTORERL) which is still at the prototype stage, the research has the character of a pilot study. A few aspects of the study which could be improved will be discussed below, and an outlook for possible extensions will be given.

[110] http://flossmetrics.org/sections/deliverables/WP1
[111] http://promisedata.org/

7 Open questions & outlook

Finally, some necessary improvements will be noted and possible extensions proposed.

7.1 Improvements for the study

7.1.1 Capturing the life cycle of issues

Issues pass through various processing phases in the issue tracking system (see section 2.2.4, p. 26), which are documented in a modification record. The data about the temporal sequence of these phases are at present not recorded by the developed issue extraction program, because it is not available like static issue data as XML files, but has to be extracted from semi-structured HTML files; the scope of this study was not sufficient for that.

7.1.2 More precise capturing of the processing time of issues

Capturing of the dynamic changes in an issue would make it possible to document more precisely the processing time of issues, in order to draw conclusions about the efficiency of the problem-solving in the software project under study. At present, the effort expended in solving an issue is greatly exaggerated, as the whole time-span from the creation of an issue to its closure is estimated. This ignores the fact that in the processing of issues, interruptions may and in fact do occur, during which no effort is expended. These interruptions are stored as events in the modification record of every issue.

7.1.3 More complete links between issues and program components

For their linking of issues to program components, Bird et al. [9:124] use the BLAME function of the version control system GIT, to establish for each line of

the program with which commit, that is, at which point in time it was originally inserted into the system. On the basis of this information, it may be possible to extend forward the time-frame for assigning issues to program components, which would lead to a more complete data basis for statistical analysis.

7.1.4 Refining the statistical research

The statistical procedures used only analyse linear correlations between the measurements. When the hypotheses were tested (see section 4.3.6, p. 73), signs of non-linear correlations were repeatedly seen in the scatter plots. With more sophisticated statistical procedures, these could be quantified and tested for significance.

Independently of this, the data should be grouped according to various individual features for testing, in order to be able to reveal correlations which remained hidden in the overall view obtained here.

7.2 Possible extensions

7.2.1 Further external measures on the basis of issue data and modification records

Only a small amount of the data available in the issue tracking system and in the modification record was used. Future research could include the other ·issue types, such as IMPROVEMENTS or TASKS. Furthermore, the data allow conclusions about the "human factor". The number of developers or their workload, measured by the number of ·issues they are assigned, or their experience, measured by the number of issues they have resolved to date, could be taken into account, in order to reach a better understanding of the interaction between the various factors.

7.2.2 Empirical comparison between functional and imperative programming

Functional programming is said to have certain advantages over imperative programming, including that it is easier to understand or "clearer" because it is expressed in terms taken from mathematics; that it is easier to test; that it is more reusable, etc. Bothe [12:16] sees the value of descriptive programming languages in general in their focus on the problem to be solved and not on the route to its solution. Ryder [80:63] believes that functional programs show less unexpected behaviour or behaviour which cannot be anticipated, as expressions have precisely

one effect and no side-effects. The advantages claimed for functional programming are generally not proven with empirical data (see section 1.2, p. 7). Solid validated software measures could serve as the basis for answering the question which Berg [6:3] posed: "[H]ow different is [functional programming] from the 'classical' imperative programming style? Is functional programming ***good*** for the development of software: can these programs be developed in a shorter time; are functional programs more reliable; are such programs easier to maintain?"

List of Figures

List of Tables

Bibliography

[1] Douglas Adams. *The Ultimate Hitchhiker's Guide: Five Complete Novels and One Story*. Gramercy Books, 2005, p. 815 (cit. on p. 27).

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullmann. *Compilerbau*. Oldenbourg, 1999 (cit. on pp. 24, 25).

[3] Herbert Amann and Joachim Escher. *Statistik III*. Birkhäuser, 2001 (cit. on p. 19).

[4] Klaus Backhaus et al. *Multivariate Analysemethoden*. 2006 (cit. on p. 9).

[5] Helmut Balzert. *Lehrbuch der Software-Technik*. Spektrum Akademischer Verlag, 1998 (cit. on pp. 19–21, 43, 64, 87).

[6] Klaas G. van den Berg. "Software Measurement and Functional Programming". PhD thesis. 1995 (cit. on pp. 7–9, 23, 63, 94).

[7] Manuel F. Bertoa, Antonio Vallecillo, and Félix García. "An Ontology for Software Measurement". In: *Ontologies for Software Engineering and Software Technology*. Springer, 2006. Chap. 6, pp. 175–196 (cit. on pp. 17, 18).

[8] Dennis Bijlsma. "Indicators of Issue Handling Efficiency". Master's thesis. 2010 (cit. on pp. 47, 71).

[9] Christian Bird et al. "Fair and Balanced? Bias in Bug-Fix Datasets". In: *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM. 2009, pp. 121–130 (cit. on pp. 34, 56, 92).

[10] Richard Bird and Philip Wadler. *Einführung in die funktionale Programmierung*. Hanser, 1992 (cit. on p. 36).

[11] Jürgen Bortz and Christof Schuster. *Statistik für Human- und Sozialwissenschaftler*. Springer, 2010 (cit. on pp. 16, 17, 19).

[12] Klaus Bothe. "Von Algol nach Java: Kontinuität und Wandel von Programmiersprachen". In: *Informatik : Aktuelle Themen im historischen Kontext*. Ed. by Wolfgang Reisig and Johann-Christoph Freytag. Springer-Verlag, 2006, pp. 1–16 (cit. on pp. 35, 37, 93).

[13] Istvan Bozó et al. "Using Impact Analysis Based Knowledge For Validating Refactoring Steps". In: *Studia Universitatis Babes-Bolyai*. Informatica 56.3 (2011) (cit. on p. 49).

[14] William J. Brown et al. *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley, 1998 (cit. on p. 22).

[15] Francesco Cesarini and Simon Thompson. *Erlang Programming*. 2009 (cit. on p. 63).

[16] Shyam R. Chidamber and Chris F. Kemerer. "A Metrics Suite for Object Oriented Design". In: *IEEE Transactions on Software Engineering* 20.6 (June 1994), pp. 476–493 (cit. on pp. 43, 44, 64, 65).

[17] Günter Clauß and Heinz Ebner. *Grundlagen der Statistik*. Berlin: Volk und Wissen, 1974 (cit. on p. 68).

[18] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. 2nd ed. Lawrence Erlbaum Associates, 1988 (cit. on p. 72).

[19] IEEE Computer Society. Standards Coordinating Committee and IEEE Standards Board. "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Standard Glossary of Software Engineering Terminology*. Ed. by Jane Radatz. Institute of Electrical and Electronics Engineers. New York, N.Y: Institute of Electrical and Electronics Engineers, 1990 (cit. on p. 13).

[20] Ely Deckers. "Verifying properties of RefactorErl-transformations using QuickCheck". URL: `http://www.ru.nl/publish/pages/578936/verifying_properties_of_refactorerl-transformations_-_ely_deckers.pdf`, accessed on 12 April 2012. Master's thesis. 2010 (cit. on p. 49).

[21] Jean-Marc Desharnais. *Analysis of ISO-IEC 9126 and 25010*. Lecture slides. URL: `http://www.cmpe.boun.edu.tr/courses/cmpe58V/fall2009/06a-Analysisof9126-2,3,4short.pdf`, accessed on 12 April 2012. 2009 (cit. on p. 15).

[22] Edsger W. Dijkstra. "EWD268 : Structured Programming". In: (Aug. 1969). URL:`http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD268.PDF`, accessed on 9 May 2012 (cit. on p. 20).

[23] DIN. *Internationales Wörterbuch der Metrologie – International Vocabulary of Basic and General Terms in Metrology*. 2nd ed. Beuth Verlag, 1994 (cit. on pp. 106–108).

[24] DIN. *Internationales Wörterbuch der Metrologie – International Vocabulary of Basic and General Terms in Metrology*. 3rd ed. Beuth Verlag, 2010 (cit. on pp. 20, 106, 108).

[25] DIN. *Software-Ergonomie : Empfehlungen für die Programmierung und Auswahl von Software*. 1st ed. (CD version). Vol. 354. DIN-Taschenbuch. Beuth Verlag, 2004 (cit. on p. 14).

[26] Klas Eriksson, Mike Williams, and Joe Armstrong. *Program Development Using Erlang - Programming Rules and Conventions*. Ericsson. Mar. 1996 (cit. on pp. 26, 63, 66).

[27] Norman E. Fenton and M. Neil. "A Critique of Software Defect Prediction Models". In: *Software Engineering, IEEE Transactions on* 25.5 (1999), pp. 675–689 (cit. on p. 21).

[28] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics : A Rigorous & Practical Approach*. 2nd ed. PWS Publishing Co., 1996 (cit. on pp. 14, 41).

[29] Norman E. Fenton, Robin W. Whitty, and Agnes A. Kaposi. "A generalised mathematical theory of structured programming". In: *Theoretical Computer Science* 36 (1985), pp. 145–171 (cit. on pp. 25, 26).

[30] Martin Fowler. *Refactoring oder: wie Sie das Design vorhandener Software verbessern*. Addison-Wesley, 2005 (cit. on p. 22).

[31] Erich Gamma et al. *Entwurfsmuster : Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 2001 (cit. on p. 22).

[32] Félix García et al. "Towards a Consistent Terminology for Software Measurement". In: *Information & Software Technology* (2006), pp. 631–644 (cit. on pp. 17, 20, 106–108).

[33] Walter Gellert et al. *Mathematik*. Kleine Enzyklopädie. Leipzig: VEB Bibliographisches Institut, 1965 (cit. on p. 19).

[34] Gerd Gigerenzer. *Messung und Modellbildung in der Psychologie*. Ernst Reinhardt GmbH & Co., 1981 (cit. on p. 23).

[35] Stephen Jay Gould. *Der falsch vermessene Mensch*. Suhrkamp Verlag, 1983, p. 400 (cit. on p. 16).

[36] Todd L. Graves et al. "Predicting fault incidence using software change history". In: *IEEE Transactions on Software Engineering* 26.7 (2000), pp. 653–661 (cit. on pp. 46, 57, 69).

[37] Jonathan L. Gross and Jay Yellen. *Handbook of Graph Theory*. Discrete mathematics and its applications. CRC, 2004 (cit. on p. 43).

[38] Rudolf Halin. *Graphentheorie*. Akademie-Verlag Berlin (licensed edition), 1989 (cit. on p. 43).

[39] T. Hall et al. "A Systematic Literature Review on Fault Prediction Performance in Software Engineering". In: () (cit. on p. 21).

[40] Maurice H. Halstead. *Elements of Software Science*. Operating and Programming Systems Series. Elsevier, 1977 (cit. on p. 8).

[41] Matthew S. Hecht. *Flow analysis of computer programs*. Programming languages series. Elsevier, 1977 (cit. on pp. 24, 25).

[42] Ilja Heitlager, Tobias Kuipers, and Joost Visser. "A practical model for measuring maintainability". In: *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*. IEEE. 2007, pp. 30–39 (cit. on pp. 14, 15).

[43] Pieter Hooimeijer and Westley Weimer. "Modeling Bug Report Quality". In: (2007), pp. 34–43 (cit. on p. 34).

[44] Tim Hopkins and Les Hatton. "Exploring defect correlations in a major Fortran numerical library". URL: `http://www.leshatton.org/Documents/NAG01_01-08.pdf`, accessed on 12 May 2012. 2008 (cit. on pp. 23, 33, 35, 62, 65, 69, 74, 81, 87–89).

[45] James W. Howatt and Albert L. Baker. "Definition and Design of a Tool for Program Control Structure Measures". In: *Proc. COMPSAC 85* 214 (1985), pp. 214–220 (cit. on p. 16).

[46] IEEE. *IEEE Standard for a Software Quality Metrics Methodology*. IEEE Standards Dept., 1998 (cit. on pp. 13, 21, 28, 72).

[47] ISO. *Information Technology : Software Product Evaluation : Quality Characteristics and Guidelines for their Use*. International Standards Organisation, 1991 (cit. on p. 14).

[48] ISO. *Software engineering : Software product Quality Requirements and Evaluation (SQuaRE) : Measurement reference model and guide*. International Standards Organisation, 2007 (cit. on p. 28).

[49] ISO. *Systems and software engineering : Systems and software Quality Requirements and Evaluation (SQuaRE) : System and software quality models*. International Standards Organisation, 2011 (cit. on pp. 14, 15, 27).

[50] JIRA. *JIRA Documentation 4.4.x*. URL: `http://confluence.atlassian.com/download/attachments/71598773/JIRA+4.4+Documentation+%28PDF%29+20110819.pdf`, accessed on 04 March 2012. JIRA. 2011 (cit. on p. 27).

[51] Elroy Jumpertz. "Using QuickCheck and Semantic Analysis to Verify Correctness of Erlang Refactoring Transformations". URL: `http://www.ru.nl/publish/pages/578936/refactoring_transformations_elroy_jumpertz.pdf`, accessed on 12 April 2012. Masterarbeit. 2010 (cit. on p. 49).

[52] Stephen H. Kan. *Metrics and Models for Software Quality Engineering*. Pearson Education Inc., 2003 (cit. on p. 48).

[53] Roland Király and Róbert Kitlei. "Application of Complexity Metrics in Functional Languages". In: *Proceedings of the 8th Joint Conference on Mathematics and Computer Science, Selected Papers*. URL: `http://www.selyeuni.sk/macs/pdf/MaCS-2010.pdf`, accessed on 12 April 2012. 2010, pp. 267–282 (cit. on pp. 7, 11, 48).

[54] Barbara A. Kitchenham, T. Dyba, and M. Jorgensen. "Evidence-based software engineering". In: *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. IEEE. 2004, pp. 273–281 (cit. on p. 28).

[55] David H. Krantz et al. *Foundations of Measurement : Additive and polynomial representation*. Vol. 1. 1991 (cit. on p. 19).

[56] Ulf Leser and Felix Naumann. *Informationsintegration*. dpunkt-Verl., 2006 (cit. on pp. 17, 54).

[57] Peter Liggesmeyer. *Software-Qualität : Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2002 (cit. on pp. 13, 19, 21, 22, 35, 43).

[58] Wolfram-Manfred Lippe. *Funktionale und Applikative Programmierung*. Springer, 2009 (cit. on pp. 37, 38).

[59] Bart J. H. Luijten. "The Influence of Software Maintainability on Issue Handling". Master's thesis. 2010 (cit. on pp. 46, 47, 55, 59).

[60] Radu Marinescu and Michele Lanza. *Object-oriented metrics in practice*. New York, 2006 (cit. on pp. 20–23, 45, 66, 67, 84).

[61] Robert C. Martin. *Agile Software Development : Principles, Patterns, and Practices*. Pearson Education Inc., 2003 (cit. on pp. 42, 44, 87).

[62] Thomas J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering*. SE 2.4 (Dec. 1976), pp. 308–320 (cit. on p. 43).

[63] Reginald N. Meeson Jr. "Functional Programming". In: *Encyclopedia of Software Engineering* 1 (1994). Ed. by John J. Marciniak, pp. 524–526 (cit. on pp. 35, 36).

[64] Beate Meffert and Olaf Hochmuth. *Werkzeuge der Signalverarbeitung*. Pearson Studium, 2004 (cit. on pp. 68, 117, 118, 121, 122).

[65] Andrew Meneely, Ben Smith, and Laurie Williams. *Software Metrics Validation Criteria: A Systematic Literature Review*. Tech. rep. 2010 (cit. on pp. 27–31, 113–116).

[66] Meine J.P. van der Meulen and Miguel A. Revilla. "Correlations between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs". In: ISSRE '07. 18th IEEE International Symposium on Software Reliability Engineering. IEEE Computer Society, 2007, pp. 203–208 (cit. on p. 68).

[67] Jan Midtgaard. "Control-flow analysis of functional programs". In: (). URL: `cs.au.dk/~jmi/Midtgaard-CSur-final.pdf`, accessed on 26 April 2012 (cit. on p. 48).

[68] Jack Moffit. *Professional XMPP*. Wiley, 2010 (cit. on p. 51).

[69] Trevor T. Moores. "Applying complexity measures to rule-based prolog programs". In: *Journal of Systems and Software* 44.1 (Dec. 1998), pp. 45–52 (cit. on p. 66).

[70] Harvey Motulsky. *Intuitive Biostatistics*. Oxford University Press, 1995 (cit. on p. 72).

[71] Randall Munroe. *xkcd : volume 0*. Breadpig Inc, 2009 (cit. on p. 62).

[72] Jerome L. Myers and Arnold Well. *Research design and statistical analysis*. 2nd ed. Vol. 1. Lawrence Erlbaum, 2003 (cit. on p. 9).

[73] Hartmut Noltemeier. *Graphentheorie : mit Algorithmen und Anwendungen*. Walter de Gruyter, 1976 (cit. on p. 43).

[74] Frank R. Oppedijk. "Comparison of the SIG Maintainability Model and the Maintainability Index". Master's thesis. 2008 (cit. on p. 15).

[75] Bernd Orth. "Grundlagen des Messens". In: *Messen und Testen* 4 (1983), pp. 136–180 (cit. on pp. 16, 19).

[76] Peter Pepper. *Funktionale Programmierung in OPAL, ML, HASKELL und GOFER*. 2nd rev. ed. Springer-Verlag, 2002 (cit. on pp. 35, 36).

[77] Marian Petre and Russel L. Winder. "Programming Languages: Models and Programming Styles". In: *Encyclopedia of Software Engineering* 2 (1994). Ed. by John J. Marciniak, pp. 892–900 (cit. on p. 35).

[78] Gustav Pomberger and Wolfgang Pree. *Software Engineering : Architektur-Design und Prozessorientierung*. 3rd ed. Hanser, 2004 (cit. on p. 42).

[79] *RefactorErl 0.9.12.01 Manual*. Eötvös Loránd University and Ericsson Hungary. 2012 (cit. on pp. 41–43, 49).

[80] Christopher Ryder. "Software Measurement for Functional Programming". PhD thesis. 2004 (cit. on pp. 6, 7, 9–11, 20, 22, 43, 55, 62, 63, 65, 68, 80, 87, 93).

[81] Christopher Ryder and Simon Thompson. "Software metrics: measuring haskell". In: *Trends in Functional Programming* (2005), pp. 31–46 (cit. on pp. 7, 9).

[82] Peter Saint-Andre, Kevin Smith, and Remko Tronçon. *XMPP: The Definitive Guide*. O'Reilly, 2009 (cit. on p. 51).

[83] Bernd-Holger Schlingloff. "Softwarequalität : Geschichte und Trends". In: *Informatik : Aktuelle Themen im historischen Kontext*. Ed. by Wolfgang Reisig and Johann-Christoph Freytag. Springer-Verlag, 2006, pp. 329–345 (cit. on pp. 20, 34).

[84] Norman F. Schneidewind. "Methodology for validating software metrics". In: *Software Engineering, IEEE Transactions on* 18.5 (1992), pp. 410–422 (cit. on p. 21).

[85] Uwe Schöning. *Theoretische Informatik : kurzgefasst*. Spektrum Akademischer Verlag, 2003 (cit. on p. 43).

[86] Diomidis Spinellis. *Code Quality*. 1st ed. Addison-Wesley, 2006 (cit. on pp. 21, 22, 65).

[87] Mate Tejfel et al. “Improving quality of software analyser and transformer tools using specification based testing”. In: *9th Joint Conference on Mathematics and Computer Science, February 9–12, 2012, Sifok, Hungary*. 2012 (cit. on p. 49).

[88] Melinda Tóth and István Bozó. “Static Analysis of Complex Software Systems Implemented in Erlang”. In: Lecture Notes in Computer Science (LNCS) 7241 (2012), pp. 451–514 (cit. on p. 48).

[89] Helge Toutenburg and Christian Heumann. *Deskriptive Statistik*. 6th ed. Springer-Verlag, 2008 (cit. on pp. 17, 67, 68, 72).

[90] Rajesh Vasa. “Growth and Change Dynamics in Open Source Software Systems”. PhD thesis. Melbourne, Australia, 2010 (cit. on p. 68).

[91] Ernest Wallmüller. *Software Quality Engineering* (cit. on p. 107).

[92] Guido Walz, ed. *Lexikon der Mathematik*. Vol. 1. Spektrum Akademischer Verlag, 2001 (cit. on p. 35).

[93] Guido Walz, ed. *Lexikon der Mathematik*. Vol. 5. Spektrum Akademischer Verlag, 2001 (cit. on p. 19).

[94] Elke Warmuth and Walter Warmuth. *Elementare Wahrscheinlichkeitsrechnung*. 1998 (cit. on p. 5).

[95] Christoph Weischer. *Sozialforschung*. UVK Verlagsgesellschaft, 2007 (cit. on p. 16).

[96] Robin W. Whitty, Norman E. Fenton, and Agnes A. Kaposi. “Structured programming: a tutorial guide”. In: *Software & Microsystems* 3.3 (1984), pp. 54–65 (cit. on p. 26).

[97] Horst Zuse. *A Framework of Software Measurement*. de Gruyter, 1998 (cit. on pp. 17, 19, 27).

[98] Horst Zuse. “Resolving the Mysteries of the Halstead Measures”. Received via e-mail from the author, 16 August 2011. 2005 (cit. on p. 19).

[99] Horst Zuse. *Software Complexity : Measures and Methods*. Vol. 4. Programming complex systems. de Gruyter, 1991 (cit. on pp. 16, 17, 19).

Appendix

A Software Measurement Ontology

The following tables are translated from García et al. [32:636ff.], with the last two columns containing the respective English term and a source for the German translation.[112]

[112] This may not be very useful in this English translation of the thesis, but was still included for reference.

[113] For the Software Measurement Ontology, García et al. [32] used the second edition of *Internationales Wörterbuch der Metrologie – International Vocabulary of Basic and General Terms in Metrology* (*VIM*) [23], which was current at the time. In the meantime, the third edition appeared [24]. For the translation of terms from the *VIM*, however, the second edition was still used here.

Table A.1 – Definitions of the terms in the SMO (I) (translation [cf. 32:636])

German term	Meta-term	Definition	Source	English term	Translation
Informationsbedarf	term	Insight necessary to manage objectives, goals, risks and problems	ISO/IEC 15939	Information need	
Messbarer Sachverhalt	term	Abstract relationship between attributes of entities and information needs	ISO/IEC 15939	Measurable concept	
Messobjekt	term	Object that is to be characterized by measuring its attributes	ISO/IEC 15939	Entity	[91:202]
Messobjektklasse	term	The collection of all entities that satisfy a given predicate	new	Entity class	
Attribut	term	A measurable physical or abstract property of an entity which is common to all entities of an entity class	following ISO/IEC 14598	Attribute	
Qualitätsmodell	term	The set of measurable characteristics and the relationships between them which provide the basis for specifying quality requirements and evaluating quality of entities of a given entity class.	following ISO/IEC 14598	Quality model	
Maß	term	A defined measurement approach with a scale. (A measurement approach is either a measurement method, a measurement function or an analysis model.)	following 14598 "metric"	Measure	
Skala	term	A set of values with defined properties.	ISO/IEC 14598	Scale	
Skalentyp	term	The nature of the relationship between values on the scale.	ISO/IEC 15939	Type of scale	
Maßeinheit	term	Quantity, defined and adopted by convention, with which any other quantity of the same kind can be compared to express the ratio of the two quantities.	VIM	Unit of measurement	[23:21][113]
Basismaß	measure	A measure of an attribute that does not depend upon a measure of any other attribute and whose measurement approach is a measurement function.	following ISO/IEC 14598 "direct metric"	base measure	
Abgeleitetes Maß	measure	A measure that is derived from other base or derived measures using a measurement function as measurement approach.	following ISO/IEC 14598 "indirect metric"	Derived measure	
Indikator	measure	A measure that is derived from other measures using an analysis model as measurement approach.	new	Indicator	

Table A.2 – Definitions of the terms in the SMO (II) (translation [cf. 32:637])

German term	Meta-term	Definition	Source	English term	Translation according to
Messmethode	Measurement approach	Logical sequence of operations, described generically, used in quantifying an attribute with respect to a specified scale. (A measurement method is the measurement approach of a base measure.)	following ISO/IEC 15939	Measurement method	[cf. 24:29]
Messfunktion	Measurement approach	Algorithm or calculation performed to combine two or more base or derived measures. (A measurement function is the measurement approach of a derived measure.)	following ISO/IEC 15939	Measurement function	[cf. 24:46]
Analysemodell	Measurement approach	Algorithm or calculation combining one or more base and/or derived measures with associated decision criteria. (An analysis model is the measurement approach of an indicator.)	following ISO/IEC 15939	Analysis model	
Entscheidungskriterium	Term	Thresholds, targets, or patterns used to determine the need for action or further investigation, or to describe the level of confidence in a given result.	ISO/IEC 15939	Decision criteria	
Messansatz	Term	Series of operation aimed at determining the value of a measurement result. (A measurement approach is either a measurement method, a measurement function, or an analysis model.)	New	Measurement approach	
Messung	Term	Totality of operations whose objective is to determine the value of a measurement result, for a given attribute of an entity, using a measurement approach.	following VIM	Measurement	following [23:31]
Messergebnis	Term	The number or category assigned to an attribute of an entity by making a measurement.	following ISO/IEC 14598 "Measure"	Measurement result	[cf. 24:30]

B Measurement and evaluation environment

B.1 Used third-party programs and libraries

- PostgreSQL 9.1.3 as database management system
- Bash 4.1.10 and 4.2.10, Ruby 1.8.7, GNU Awk 3.1.8, Perl 5.12.4
- TRANG: `http://www.thaiopensource.com/relaxng/trang.html`

B.2 Scripts and programs

The measurement environment consists of a collection of scripts and programs, which were developed for this study. They are contained in the electronic appendix to this thesis in the following directory/file structure:

- measurement environment
  - 1 setup
    - README.txt
  - 2 database
    - createdb.sh
    - createdb.sql
  - 3a import_todo
    - (ten sub-directories with the raw measurement data)
  - 4 importissues
    - ejab-issues
    - importissues.pl
  - 5 importlog
    - ejabberd_log_120317.txt
    - importlog.rb
  - 6 linecount
    - linecount.rb
    - linediff.rb
  - 7 cleanup
    - purgedb1.sql

        - purgedb2.sql
        - purgedb3.sql
        - unify_names.sql
      - 8 readpatch
        - ejabberd_revisions_until_2011-12-30.txt
        - fixlinks.sql
        - issuecodelink.rb
      - 9 indirect measures
        - create_zeros.sql
        - fill_measures.sql
        - get_indirect_measures1.sql
        - get_indirect_measures2.sql
        - get_indirect_measures3.sql
      - 10 export
        - checkmeasures.r
        - describemeasures.r
        - exportmeasures.rb
        - measures.r
  - importmeasures.rb
  - utils.rb

B.2.1 Scripts for evaluating tool validity

B.2.1.1 Number of modules

Listing B.1 – Bash script for printing the number of Erlang files per ejabberd release

```
#!/usr/bin/env bash

# get number of Erlang source files for all subdirectories (
    non-recursively)

for d in `find . -maxdepth 1 -type d | sort`
do
  echo "$d: `find $d -name "*.erl" -print | wc`"
done
```

Listing B.2 – SQL query to get the number of loaded modules per ejabberd release

```
select m.revision, count(distinct m.mod)
from dmeasure m, m_loaded l
where m.revision like 'r%'
      and l.revision = m.revision
      and l.mod = m.mod
```

```
-       and l.loaded = true
- group by m.revision
- order by m.revision
```

B.2.1.2 Number of functions

Listing B.3 – SQL query to get the number of functions that were imported into the measurement environment

```
1 select m.revision, count(distinct (m.mod,m.fun,m.arity))
- from dmeasure m
- where m.revision like 'r%'
-       and (m.revision, m.mod, m.fun, m.arity) not in
5          (select nd.revision, nd.mod, nd.fun, nd.arity
-           from mf_notdefined nd
-           where nd.revision = m.revision)
- group by m.revision
- order by m.revision
```

Listing B.4 – SQL query to get the measurement values for the number of functions per EJABBERD release

```
1 select revision, sum(value)
- from dmeasure
- where revision like 'r%' and measure = 'number_of_fun'
- group by revision
5 order by revision
```

B.2.1.3 Number of lines

Listing B.5 – RUBY script to count lines. Excerpt – full version available in the electronic appendix, `code/6 linecount/linecount.rb`

```
1 while (line = file.gets)
-   if line.match(/^$/) then blanks += 1 end
-   if line.match(/^[ \t]+$/) then quasiblanks += 1 end
-   if line.match(/\S+/) then loc += 1 end
5   if line.match(/^\s*%/) then cloc += 1 end
-   if line.match(/\w+%/) then pcloc += 1 end
- end
```

B.2.2 Scripts for linking issues to code

Listing B.6 – RUBY script for analysing PATCH files. Excerpt – full version available in the electronic appendix, `code/8 readpatch/issuecodelink.rb`

```
1 # <id> is just used as source identifier
- def getPatchTarget(ikey, revID, id, filename, note)
-   @log.info("read patch for #{revID}: #{filename}...\n")
```

```
-
5   patchfile = File.new(filename, "r")
-   while (line = patchfile.gets)
-     if m = line.match(/^\-\-\- (\S+)/)
-       mod = File.basename(m[1], ".erl")
-     elsif m = line.match(/^@@ \-(\d+),?(\d*).*\+(\d+),?(\d*)
        .* @@/)
10      startindex, endindex = 1, 2
-       startline = m[startindex].to_i()
-       endline = m[startindex].to_i() + m[endindex].to_i() - 1
-
-       targetFuns = getFunctions(revID, mod, startline,
          endline)
15      print "FUNS: #{targetFuns.to_s()}\n"
-       targetFuns.each { |r, m, f, a|
-         addIssueCodeLink(ikey, revID, m, f, a, "#{note},#{id}
            /#{filename}")
-       }
-     end
20  end
-   patchfile.close
- end
```

C Supplemental material

C.1 Validation criteria that were not examined

"Actionability" and constructivity, #2,#11 A measure should be suitable as the basis for decisions by project managers ("*actionability*") [65:15], or it should help researchers in understanding software quality (*constructivity*) [65:17]. "Actionability" mixes the characteristic of a measure of correctly representing a relevant quality feature with the general ability of a manager to decide rationally. The latter cannot be examined in the framework of this study. ·Constructivity appears to be too abstract to be testable. It is concretized by ·association and ·discriminative power, among others.

Suitable granularity, #4 A measure should not be too finely or too coarsely grained [65:15]. This is not examined here, because it likely depends heavily on the needs of individual users what is "too fine" and what is "too coarse".

Attribute validity, #6 Measurement values have to correctly represent the property to be measured [65:16]. This is just another formulation of the definition for ·internal validity.

Causal model validity and causal relationship validity, #7,#8 These criteria are related to the usefulness of a measure as part of a predictive model (*causal model validity*), or, respectively, whether even a causal relationship with an external quality feature can be proven (*causal relationship validity*) [65:16]. Examining predictive models would exceed the framework of the present study. Besides, the distinction between the two criteria is not clear.

Content validity, #9 A measure should span the investigated quality dimension completely [65:16].

Economic productivity and usability, #15/#47 The effort for performing the measurement should not exceed the benefit (*economic productivity*) [65:17]. This criterion is classified by Meneely, Smith, and Williams [65:24] as a form of internal, theoretical validity. However, the costs of measurement can only be determined empirically and moreover depend on the efficiency of the implementation, and the benefit is an external phenomenon – therefore, a classification as external, empirical validity, or construct validity, respectively, appears to be more sensible. Usability (#47) as defined by Meneely, Smith, and Williams [65:23] is synonymous to economic productivity.

Monotonous increase, #21 The combined measurement value of two components must never be smaller than the individual measurement values of the components [65:18]. By generalizing ***one*** intuitive concept of software measures to properties of a possibly very different kind, this criterion unnecessarily restricts for all measures the characteristic of being a homomorphism.

Interaction sensitivity, #22 Different kinds of interaction (or combination) of two components should lead to different measurement values [65:18]. Because it is internal and theoretic, this criterion can be examined separately and is therefore passed over here.

Internal consistency, #23 The components of a ·derived measure should capture the same property and should be related [65:19]. This property is the opposite of ·factor independence and an unnecessary intensification of ·dimension consistency. According to this criterion, the quotient of distance and time would not be a valid (speed) measure, since two different properties are combined.

Monotonicity, #25 The combination of two components must not result in a lower measurement value than the individual components [65:19]. See ·monotonous increase.

Reliability, #26 The measurement should be precise and reproducible [65:19]. This is a summary of ·internal consistency and ·stability.

Non-colinearity, #27 The correlation of an internal measure with an external measure should be preserved when other influences are controlled for [65:19]. This is a special case of ·improvement validity.

Safety against abuse, #28 It must not be possible to manipulate measurement values by applying irrelevant changes on the measurement object [65:19]. This property follows from ·internal validity, but does not offer a concrete approach for its verification.

Non-uniformity, #29 For two different measurement objects, a measure should also give different measurement values [65:19]. Insofar as the measurement objects should be different ***with regard to the measured property***, this criterion is just another formulation of the characteristic of a ·measure to be a homomorphism. Apart from that, it is definitely possible that different measurement objects are equivalent in individual aspects, with correspondingly equal measurement values.

Permutation validity, #31 Measurement values should be influenced by changes in the order of program instructions [65:31]. This criterion was proposed for the pseudo-property "complexity" (see section 2.1.1, p. 15). A general applicability is not evident.

Suitability for predictions and as part of a predictive system, #32,#33 An external quality property should be predicted with sufficient precision by the measure itself (#32) or as part of a predictive model (#33), respectively [65:20]. Predictive models are not examined in the present work.

Process or product relevance, #34 Through adaptation, a measure should stay valid even in a new area of application [65:20]. This property is examined by the above mentioned continuous validation.

Rank consistency, #36 A measure should lead to the same ranking as an external quality property (or its measure, respectively) [65:20]. This is a special case of ·association or, respectively, ·trackability.

Transformation invariance and robustness against renaming, #44,#37 A measurement value should not be influenced through transformation (#44) of the measurement object, especially through renaming of identifiers (#37) [65:20,22]. This is a peculiar view of ·internal validity which is based on the unproven assumption that the names of identifiers (#37) or, even more daring, the structure of the software (#44) are irrelevant for the software quality. ·Robustness against renaming is a special case of ·transformation invariance.

Representation condition, #39 *Representation condition* denotes the characteristic of a ·measure of being a homomorphism [65:21]. Meneely, Smith, and Williams [65:21] claim that this means that ***every*** property of the formal relational system must have an equivalent in the empirical relational system. However, measurement is about representing empirical relations through corresponding formal relations. Depending on ***which*** empirical relations are preserved in the measurement values, the measure has a different ·scale type (see section 2.1.2.1, p. 17).

Unit validity, #46 The unit of measurement should be appropriate for the measured property [65:22]. Insofar as this is to say that the measurement values in this unit represent the empirical relations, this is just another formulation for ·internal validity. See also ·granularity.

D Supplementary tables

D.1 For section 4.3.3, p. 68 and section 4.3.6, p. 73

Table D.1 – Central moments of the internal measures for functions. $\mathfrak{z}_2$ is the dispersion, $\mathfrak{z}_3$ the skew and $\mathfrak{z}_4$ the kurtosis [64:65].

Maß	$\mathfrak{z}_0$	$\mathfrak{z}_1$	$\mathfrak{z}_2$	$\mathfrak{z}_3$	$\mathfrak{z}_4$
LOC_F	1.0	0.0	1.0	7.3	84.5
MaxCall_F	1.0	-0.0	1.0	2.0	7.1
MaxCase_F	1.0	-0.0	1.0	1.9	7.5
MaxNest_F	1.0	0.0	1.0	1.6	5.8
$\text{NumClauses}_\text{F}$	1.0	-0.0	1.0	9.8	124.6
$\text{RecBranch}_\text{F}$	1.0	-0.0	1.0	32.6	1435.7
FanIn_F	1.0	0.0	1.0	25.9	957.3
FanOut_F	1.0	0.0	1.0	3.8	30.1
NumAnon_F	1.0	-0.0	1.0	5.7	59.1
NumSend_F	1.0	-0.0	1.0	20.8	583.6
Returns_F	1.0	0.0	1.0	9.8	160.1
Cyc_F	1.0	0.0	1.0	8.8	120.7

Table D.2 – Central moments of the internal measures for modules. $\mathfrak{z}_2$ is the dispersion, $\mathfrak{z}_3$ the skew and $\mathfrak{z}_4$ the kurtosis [64:65].

Maß	$\mathfrak{z}_0$	$\mathfrak{z}_1$	$\mathfrak{z}_2$	$\mathfrak{z}_3$	$\mathfrak{z}_4$
NCLOC_M	1.0	-0.0	1.0	3.6	19.1
CLOC_M	1.0	0.0	1.0	4.3	27.7
LOC_M	1.0	0.0	1.0	3.5	18.6
LOCF_M	1.0	-0.0	1.0	3.6	18.7
Included_M	1.0	-0.0	1.0	0.3	2.6
Imported_M	1.0	0.0	1.0	7.6	59.2
NumMac_M	1.0	-0.0	1.0	3.0	12.9
NumRec_M	1.0	0.0	1.0	2.4	12.0
NumFun_M	1.0	0.0	1.0	2.9	15.4
CallsIn_M	1.0	-0.0	1.0	8.4	89.6
CallsOut_M	1.0	0.0	1.0	2.8	12.9
Calls_M	1.0	-0.0	1.0	3.8	22.8
$\text{NumClauses}_\text{M}$	1.0	0.0	1.0	2.8	12.4
Returns_M	1.0	-0.0	1.0	3.3	17.1
NumAnon_M	1.0	-0.0	1.0	5.9	49.3
NumSend_M	1.0	0.0	1.0	5.9	49.5
$\text{RecBranch}_\text{M}$	1.0	-0.0	1.0	5.3	40.2
MaxCall_M	1.0	-0.0	1.0	0.9	3.2
MaxCase_M	1.0	-0.0	1.0	0.4	2.6
MaxNest_M	1.0	-0.0	1.0	0.3	2.7
$\text{NumNonRec}_\text{M}$	1.0	-0.0	1.0	2.8	14.8
$\text{NumTrivRec}_\text{M}$	1.0	0.0	1.0	3.3	16.6
$\text{NumNonTrivRec}_\text{M}$	1.0	0.0	1.0	0.6	2.3
WMC(CYC)_M	1.0	-0.0	1.0	3.7	21.1
RFC_M	1.0	-0.0	1.0	2.9	14.2
AvgLOC_M	1.0	0.0	1.0	2.5	12.9
AvgCyc_M	1.0	-0.0	1.0	2.2	12.1
AvgCalls_M	1.0	0.0	1.0	3.3	20.2
AvgOut_M	1.0	0.0	1.0	-1.4	3.6
CBO_M	1.0	0.0	1.0	3.7	27.4
InstF_M	1.0	0.0	1.0	-1.0	2.9
InstM_M	1.0	0.0	1.0	-0.6	2.7

Table D.3 – Some characteristics of the internal measures for functions. $x_{\min}$ and $x_{\max}$ are the smallest and largest measurement value, respectively, $\tilde{d}_{0,5}$ is the mean absolute deviation from the median. $x_{0,25}$, $x_{0,5}$ und $x_{0,75}$ are lower quartile, median and upper quartile. Values up to x_{lo} and from x_{hi} are considered "outliers" (see section 4.3.2.1, p. 67)

Maß	$x_{\min}$	$x_{\max}$	$\tilde{d}_{0.5}$	$\tilde{x}_{\text{lo}}$	$\tilde{x}_{0.25}$	$\tilde{x}_{0.5}$	$\tilde{x}_{0.75}$	$\tilde{x}_{\text{hi}}$
LOC_{F}	1.0	504.0	11.9	-17.0	4.0	9.0	18.0	39.0
$\text{MaxCall}_{\text{F}}$	0.0	30.0	3.2	-6.5	1.0	2.0	6.0	13.5
$\text{MaxCase}_{\text{F}}$	0.0	6.0	0.6	-1.5	0.0	0.0	1.0	2.5
$\text{MaxNest}_{\text{F}}$	0.0	8.0	0.9	-1.5	0.0	0.0	1.0	2.5
$\text{NumClauses}_{\text{F}}$	1.0	41.0	0.6	1.0	1.0	1.0	1.0	1.0
$\text{RecBranch}_{\text{F}}$	0.0	87.0	0.2	0.0	0.0	0.0	0.0	0.0
FanIn_{F}	0.0	761.0	2.6	-3.0	0.0	1.0	2.0	5.0
FanOut_{F}	0.0	58.0	2.3	-3.5	1.0	2.0	4.0	8.5
$\text{NumAnon}_{\text{F}}$	0.0	15.0	0.3	0.0	0.0	0.0	0.0	0.0
$\text{NumSend}_{\text{F}}$	0.0	10.0	0.0	0.0	0.0	0.0	0.0	0.0
$\text{Returns}_{\text{F}}$	1.0	112.0	1.5	-2.0	1.0	1.0	3.0	6.0
Cyc_{F}	1.0	112.0	2.0	-2.0	1.0	2.0	3.0	6.0

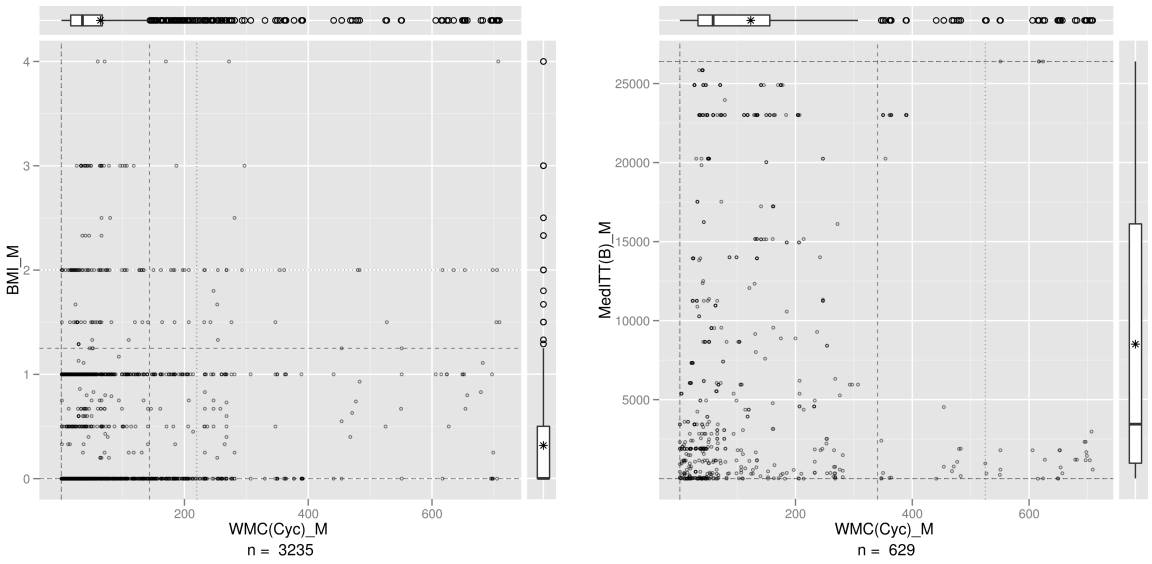


Figure D.1 – Scatter plots for $\text{WMC(Cyc)}_{\text{M}}$ and BMI_{M} (left), and $\text{MedITT(B)}_{\text{M}}$ (right)

Table D.4 – Some characteristics of the internal measures for modules. $x_{\min}$ and $x_{\max}$ are the smallest and largest measurement values, respectively, $\tilde{d}_{0,5}$ is the mean absolute deviation from the median. $x_{0,25}$, $x_{0,5}$ and $x_{0,75}$ are lower quartile, median and upper quartile. Values up to x_{lo} and from x_{hi} are considered "outliers" (see section 4.3.2.1, p. 67)

Maß	$x_{\min}$	$x_{\max}$	$\tilde{d}_{0.5}$	$\tilde{x}_{\text{lo}}$	$\tilde{x}_{0.25}$	$\tilde{x}_{0.5}$	$\tilde{x}_{0.75}$	$\tilde{x}_{\text{hi}}$
NCLOC_M	4.0	3699.0	238.6	-349.5	57.0	148.0	328.0	734.5
CLOC_M	0.0	543.0	32.1	-41.5	20.0	32.0	61.0	122.5
LOC_M	15.0	3862.0	273.2	-352.5	120.0	202.0	435.0	907.5
LOCF_M	4.0	3724.0	256.7	-367.5	84.0	163.0	385.0	836.5
Included_M	0.0	5.0	1.0	-0.5	1.0	2.0	2.0	3.5
Imported_M	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
NumMac_M	0.0	26.0	2.4	-4.5	0.0	0.0	3.0	7.5
NumRec_M	0.0	8.0	0.7	-1.5	0.0	0.0	1.0	2.5
NumFun_M	1.0	163.0	13.5	-21.0	9.0	16.0	29.0	59.0
CallsIn_M	0.0	934.0	18.6	-12.0	0.0	2.0	8.0	20.0
CallsOut_M	0.0	402.0	34.5	-63.0	12.0	30.0	62.0	137.0
Calls_M	0.0	1070.0	64.3	-101.5	23.0	45.0	106.0	230.5
$\text{NumClauses}_\text{M}$	1.0	271.0	23.3	-30.5	10.0	21.0	37.0	77.5
Returns_M	1.0	486.0	37.0	-46.0	14.0	32.0	54.0	114.0
NumAnon_M	0.0	126.0	5.4	-10.5	0.0	2.0	7.0	17.5
NumSend_M	0.0	20.0	0.7	0.0	0.0	0.0	0.0	0.0
$\text{RecBranch}_\text{M}$	0.0	101.0	4.0	-6.0	0.0	1.0	4.0	10.0
MaxCall_M	0.0	30.0	5.1	-8.0	4.0	7.0	12.0	24.0
MaxCase_M	0.0	6.0	1.2	-2.0	1.0	2.0	3.0	6.0
MaxNest_M	0.0	8.0	1.3	-1.0	2.0	3.0	4.0	7.0
$\text{NumNonRec}_\text{M}$	1.0	143.0	12.3	-21.5	7.0	14.0	26.0	54.5
$\text{NumTrivRec}_\text{M}$	1.0	20.0	1.8	-3.5	1.0	2.0	4.0	8.5
$\text{NumNonTrivRec}_\text{M}$	1.0	5.0	0.9	0.5	2.0	2.0	3.0	4.5
WMC(CYC)_M	1.0	709.0	47.6	-60.5	16.0	35.0	67.0	143.5
RFC_M	1.0	548.0	46.4	-77.0	22.0	45.0	88.0	187.0
AvgLOC_M	0.1	86.5	7.0	-7.2	8.2	12.1	18.5	33.9
AvgCyc_M	0.0	13.0	1.0	-0.7	1.7	2.2	3.2	5.6
AvgCalls_M	0.0	25.7	1.6	-1.6	1.6	2.6	3.7	6.8
AvgOut_M	0.0	1.0	0.2	0.2	0.7	0.9	1.0	1.5
CBO_M	0.0	174.0	9.5	-24.0	0.0	8.0	16.0	40.0
InstF_M	0.0	1.0	0.2	-0.1	0.6	0.9	1.0	1.6
InstM_M	0.0	1.0	0.2	-0.1	0.5	0.7	0.9	1.5

Table D.5 – Central moments of the external measures for functions. $\mathfrak{z}_2$ is the dispersion, $\mathfrak{z}_3$ is the skew and $\mathfrak{z}_4$ is the kurtosis [64:65].

Maß	$\mathfrak{z}_0$	$\mathfrak{z}_1$	$\mathfrak{z}_2$	$\mathfrak{z}_3$	$\mathfrak{z}_4$
NumIss(B)$_F$	1.0	-0.0	1.0	6.6	62.6
NumIss(N)$_F$	1.0	0.0	1.0	4.5	26.2
NumIss(T)$_F$	1.0	-0.0	1.0	14.8	228.4
NumIss(I)$_F$	1.0	0.0	1.0	3.8	24.0
StartIss(B)$_F$	1.0	-0.0	1.0	9.0	116.7
StartIss(N)$_F$	1.0	0.0	1.0	8.8	88.5
StartIss(T)$_F$	1.0	0.0	1.0	19.1	365.9
StartIss(I)$_F$	1.0	-0.0	1.0	7.5	72.8
EndIss(B)$_F$	1.0	-0.0	1.0	10.7	154.5
EndIss(N)$_F$	1.0	0.0	1.0	10.2	118.8
EndIss(T)$_F$	1.0	0.0	1.0	25.7	661.6
EndIss(I)$_F$	1.0	0.0	1.0	7.8	79.5
MedITT$_F$	1.0	0.0	1.0	0.7	2.1
MedITT(B)$_F$	1.0	-0.0	1.0	0.8	1.9
MedITT(N)$_F$	1.0	0.0	1.0	0.3	2.1
MedITT(T)$_F$	1.0	-0.0	1.0	1.6	3.8
MedITT(I)$_F$	1.0	0.0	1.0	0.7	1.8
StartRate(B)$_F$	1.0	-0.0	1.0	10.5	139.8
StartRate(N)$_F$	1.0	-0.0	1.0	16.1	368.4
StartRate(T)$_F$	1.0	-0.0	1.0	36.7	1659.6
StartRate(I)$_F$	1.0	0.0	1.0	12.2	201.8
EndRate(B)$_F$	1.0	0.0	1.0	17.5	505.4
EndRate(N)$_F$	1.0	0.0	1.0	19.1	459.7
EndRate(T)$_F$	1.0	-0.0	1.0	45.2	2267.2
EndRate(I)$_F$	1.0	0.0	1.0	9.0	105.2
ImproveRate$_F$	1.0	-0.0	1.0	5.0	26.4
BMI$_F$	1.0	0.0	1.0	5.3	37.2
BugRate$_F$	1.0	0.0	1.0	4.6	23.1

Table D.6 – Central moments of the external measures for modules. $\mathfrak{z}_2$ is the dispersion, $\mathfrak{z}_3$ is the skew and $\mathfrak{z}_4$ is the kurtosis [64:65].

Maß	$\mathfrak{z}_0$	$\mathfrak{z}_1$	$\mathfrak{z}_2$	$\mathfrak{z}_3$	$\mathfrak{z}_4$
NumIss(B)$_M$	1.0	0.0	1.0	6.6	78.7
NumIss(N)$_M$	1.0	0.0	1.0	3.1	15.0
NumIss(T)$_M$	1.0	0.0	1.0	2.2	7.5
NumIss(I)$_M$	1.0	0.0	1.0	4.0	27.2
StartIss(B)$_M$	1.0	-0.0	1.0	10.2	183.0
StartIss(T)$_M$	1.0	-0.0	1.0	2.1	6.6
StartIss(I)$_M$	1.0	0.0	1.0	5.9	57.5
EndIss(B)$_M$	1.0	0.0	1.0	9.4	163.5
EndIss(N)$_M$	1.0	0.0	1.0	5.8	54.5
EndIss(T)$_M$	1.0	-0.0	1.0	1.9	4.9
EndIss(I)$_M$	1.0	0.0	1.0	4.8	30.7
MedITT$_M$	1.0	0.0	1.0	0.9	2.9
MedITT(B)$_M$	1.0	0.0	1.0	0.8	2.1
MedITT(N)$_M$	1.0	-0.0	1.0	0.3	2.0
MedITT(T)$_M$	1.0	0.0	1.0	2.0	5.8
MedITT(I)$_M$	1.0	-0.0	1.0	0.6	2.1
StartRate(B)$_M$	1.0	-0.0	1.0	6.8	75.9
StartRate(N)$_M$	1.0	-0.0	1.0	6.1	51.8
StartRate(T)$_M$	1.0	0.0	1.0	3.3	13.3
StartRate(I)$_M$	1.0	0.0	1.0	5.3	38.6
EndRate(B)$_M$	1.0	-0.0	1.0	7.1	79.2
EndRate(N)$_M$	1.0	-0.0	1.0	6.9	64.9
EndRate(T)$_M$	1.0	-0.0	1.0	3.5	14.3
EndRate(I)$_M$	1.0	-0.0	1.0	5.2	36.9
ImproveRate$_M$	1.0	-0.0	1.0	2.6	8.3
BMI$_M$	1.0	-0.0	1.0	2.1	8.4
BugRate$_M$	1.0	0.0	1.0	2.4	7.8

Table D.7 – Some characteristics of the external measures for functions. $x_{\min}$ and $x_{\max}$ are the smallest and largest measurement values, respectively, $\tilde{d}_{0,5}$ is the mean absolute deviation from the median. $x_{0,25}$, $x_{0,5}$ and $x_{0,75}$ are the lower quartile, median and upper quartile. Values up to x_{lo} and from x_{hi} are considered "outliers" (see section 4.3.2.1, p. 67)

Maß	$x_{\min}$	$x_{\max}$	$\tilde{d}_{0.5}$	$\tilde{x}_{\text{lo}}$	$\tilde{x}_{0.25}$	$\tilde{x}_{0.5}$	$\tilde{x}_{0.75}$	$\tilde{x}_{\text{hi}}$
NumIss(B)$_{\text{F}}$	0.0	7.0	0.1	0.0	0.0	0.0	0.0	0.0
NumIss(N)$_{\text{F}}$	1.0	4.0	0.1	1.0	1.0	1.0	1.0	1.0
NumIss(T)$_{\text{F}}$	0.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0
NumIss(I)$_{\text{F}}$	0.0	7.0	0.1	0.0	0.0	0.0	0.0	0.0
StartIss(B)$_{\text{F}}$	0.0	5.0	0.0	0.0	0.0	0.0	0.0	0.0
StartIss(N)$_{\text{F}}$	0.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0
StartIss(T)$_{\text{F}}$	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
StartIss(I)$_{\text{F}}$	0.0	4.0	0.0	0.0	0.0	0.0	0.0	0.0
EndIss(B)$_{\text{F}}$	0.0	6.0	0.0	0.0	0.0	0.0	0.0	0.0
EndIss(N)$_{\text{F}}$	0.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0
EndIss(T)$_{\text{F}}$	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
EndIss(I)$_{\text{F}}$	0.0	4.0	0.0	0.0	0.0	0.0	0.0	0.0
MedITT$_{\text{F}}$	0.1	38591.9	9592.3	-24190.0	4128.4	9830.8	23007.3	51325.7
MedITT(B)$_{\text{F}}$	0.1	26396.7	7255.9	-19831.1	1166.1	3921.1	15164.2	36161.3
MedITT(N)$_{\text{F}}$	0.2	34340.6	8399.9	-19561.6	3683.2	14613.0	19179.8	42424.6
MedITT(T)$_{\text{F}}$	0.2	13450.6	2574.7	-2336.3	47.9	47.9	1637.3	4021.5
MedITT(I)$_{\text{F}}$	0.1	38591.9	10120.1	-28692.3	5022.8	9830.8	27499.5	61214.6
StartRate(B)$_{\text{F}}$	0.0	1.5	0.0	0.0	0.0	0.0	0.0	0.0
StartRate(N)$_{\text{F}}$	0.0	1.5	0.0	0.0	0.0	0.0	0.0	0.0
StartRate(T)$_{\text{F}}$	0.0	1.5	0.0	0.0	0.0	0.0	0.0	0.0
StartRate(I)$_{\text{F}}$	0.0	1.5	0.0	0.0	0.0	0.0	0.0	0.0
EndRate(B)$_{\text{F}}$	0.0	2.9	0.0	0.0	0.0	0.0	0.0	0.0
EndRate(N)$_{\text{F}}$	0.0	1.5	0.0	0.0	0.0	0.0	0.0	0.0
EndRate(T)$_{\text{F}}$	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
EndRate(I)$_{\text{F}}$	0.0	1.5	0.0	0.0	0.0	0.0	0.0	0.0
ImproveRate$_{\text{F}}$	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
BMI$_{\text{F}}$	0.0	4.0	0.1	0.0	0.0	0.0	0.0	0.0
BugRate$_{\text{F}}$	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0

Table D.8 – Some characteristics of the external measures for modules. $x_{\min}$ and $x_{\max}$ are the smallest and the largest measurement value, respectively, $\tilde{d}_{0,5}$ is the mean absolute deviation from the median. $x_{0,25}$, $x_{0,5}$ and $x_{0,75}$ are the lower quartile, median and upper quartile. Values up to x_{lo} and from x_{hi} are considered as "outliers" (see section 4.3.2.1, p. 67)

Maß	$x_{\min}$	$x_{\max}$	$\tilde{d}_{0.5}$	$\tilde{x}_{\text{lo}}$	$\tilde{x}_{0.25}$	$\tilde{x}_{0.5}$	$\tilde{x}_{0.75}$	$\tilde{x}_{\text{hi}}$
NumIss(B)$_{\text{M}}$	0.0	22.0	0.4	0.0	0.0	0.0	0.0	0.0
NumIss(N)$_{\text{M}}$	1.0	9.0	0.5	-0.5	1.0	1.0	2.0	3.5
NumIss(T)$_{\text{M}}$	0.0	3.0	0.2	0.0	0.0	0.0	0.0	0.0
NumIss(I)$_{\text{M}}$	0.0	14.0	0.5	-1.5	0.0	0.0	1.0	2.5
StartIss(B)$_{\text{M}}$	0.0	18.0	0.2	0.0	0.0	0.0	0.0	0.0
StartIss(T)$_{\text{M}}$	0.0	2.0	0.2	0.0	0.0	0.0	0.0	0.0
StartIss(I)$_{\text{M}}$	0.0	9.0	0.1	0.0	0.0	0.0	0.0	0.0
EndIss(B)$_{\text{M}}$	0.0	18.0	0.2	0.0	0.0	0.0	0.0	0.0
EndIss(N)$_{\text{M}}$	0.0	6.0	0.1	0.0	0.0	0.0	0.0	0.0
EndIss(T)$_{\text{M}}$	0.0	2.0	0.2	0.0	0.0	0.0	0.0	0.0
EndIss(I)$_{\text{M}}$	0.0	5.0	0.1	0.0	0.0	0.0	0.0	0.0
MedITT$_{\text{M}}$	0.0	38591.9	8117.4	-22901.5	648.9	6417.8	16349.1	39899.5
MedITT(B)$_{\text{M}}$	0.0	26396.7	7110.0	-21099.4	645.5	3265.8	15142.1	36887.0
MedITT(N)$_{\text{M}}$	0.2	34340.6	7882.3	-19561.6	3683.2	12853.8	19179.8	42424.6
MedITT(T)$_{\text{M}}$	0.1	13450.6	1490.2	-208.5	9.2	72.7	154.4	372.2
MedITT(I)$_{\text{M}}$	0.1	38591.9	9394.9	-25833.9	5022.8	10028.9	25594.0	56450.7
StartRate(B)$_{\text{M}}$	0.0	3.6	0.1	0.0	0.0	0.0	0.0	0.0
StartRate(N)$_{\text{M}}$	0.0	1.5	0.0	0.0	0.0	0.0	0.0	0.0
StartRate(T)$_{\text{M}}$	0.0	1.5	0.1	0.0	0.0	0.0	0.0	0.0
StartRate(I)$_{\text{M}}$	0.0	1.5	0.0	0.0	0.0	0.0	0.0	0.0
EndRate(B)$_{\text{M}}$	0.0	3.9	0.1	0.0	0.0	0.0	0.0	0.0
EndRate(N)$_{\text{M}}$	0.0	1.9	0.0	0.0	0.0	0.0	0.0	0.0
EndRate(T)$_{\text{M}}$	0.0	1.0	0.1	0.0	0.0	0.0	0.0	0.0
EndRate(I)$_{\text{M}}$	0.0	1.6	0.0	0.0	0.0	0.0	0.0	0.0
ImproveRate$_{\text{M}}$	0.0	1.0	0.1	0.0	0.0	0.0	0.0	0.0
BMI$_{\text{M}}$	0.0	4.0	0.3	-1.2	0.0	0.0	0.8	2.1
BugRate$_{\text{M}}$	0.0	1.0	0.1	0.0	0.0	0.0	0.0	0.0

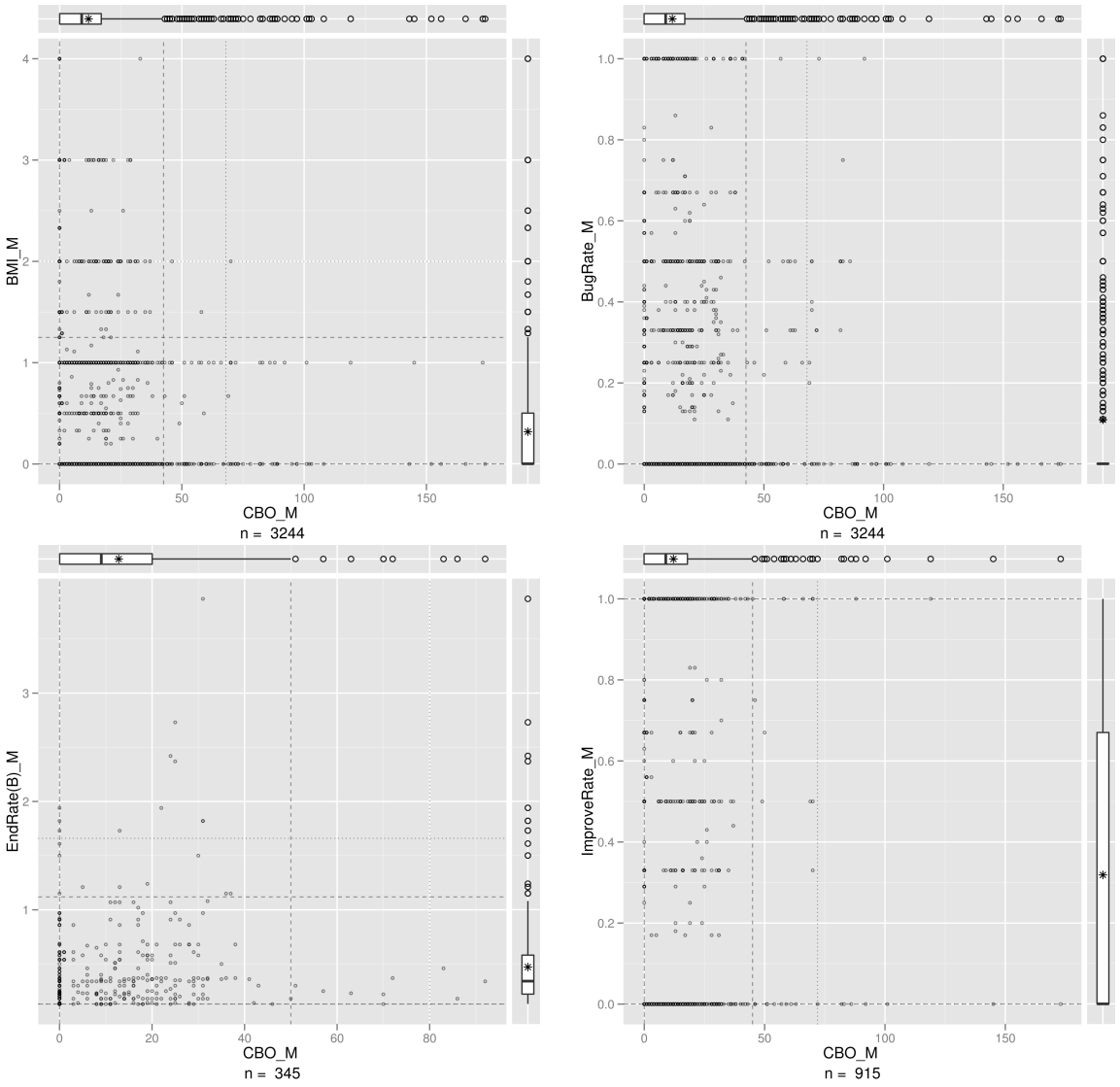


Figure D.2 – Scatter plots for the measure $\mathrm{CBO_M}$ with the measures (from left to right and from top to bottom): $\mathrm{BMI_M}$, $\mathrm{BugRate_M}$, $\mathrm{EndRate(B)_M}$, $\mathrm{ImproveRate_M}$

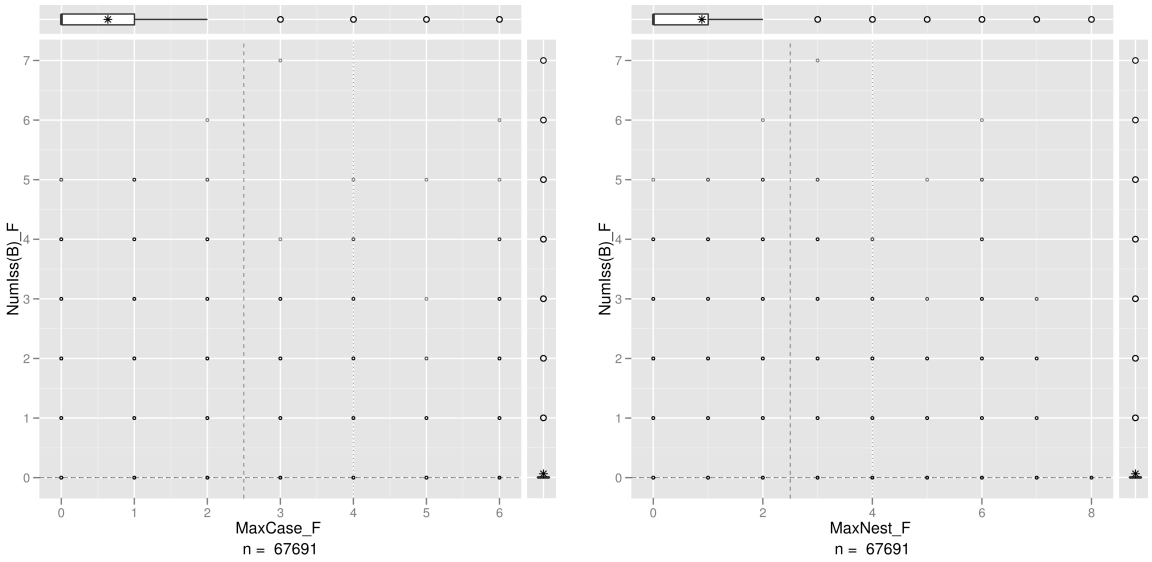


Figure D.3 – Scatter plots for MaxCase$_F$ and NumIss(B)$_F$ (left) as well as MaxNest$_F$ and NumIss(B)$_F$ (right)

Index

Statement of authorship

I hereby declare that I authored the present thesis on my own and only using the listed sources and supporting material. Furthermore, I declare that this is the first time I am submitting a Diploma thesis in this field of study.

Berlin, 12 May 2012 .....